U.S.N.A. --- Trident Scholar project report; no. 314 (2003)

A STUDY OF MAGNETICALLY-COUPLED 2-DIMENSIONAL RECONFIGUABLE MODULAR ROBOTS

By

Midshipman Sean A. Patterson, Class of 2003
United States Naval Academy
Annapolis, Maryland

____
(signature)

Certification of Advisors Approval

Professor Kenneth A. Knowles
Department of Weapons and Systems Engineering

____
(signature)

____
(date)

Associate Professor Bradley E. Bishop
Department of Weapons and Systems Engineering

____
(signature)

____
(date)

Acceptance of Trident Scholar Committee

Professor Joyce E. Shade
Deputy Director of Research & Scholarship

____
(signature)

____
(date)

USNA-1531-2

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **06 MAY 2003**	2. REPORT TYPE **N/A**	3. DATES COVERED **-**
4. TITLE AND SUBTITLE **A Study Of Magnetically-Coupled 2-Dimensional Reconfiguable Modular Robots**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **United States Naval Academy Annapolis, Maryland**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR’S ACRONYM(S)
		11. SPONSOR/MONITOR’S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release, distribution unlimited**		
13. SUPPLEMENTARY NOTES **The original document contains color images.**		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**UU**	**68**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Abstract

This Trident Scholar project consisted of the development and investigation of identical modular autonomous robots that used only active magnetic forces to self-reconfigure and passive magnetic forces to maintain themselves in a network of independent modular robots. The resultant modular group configurations have the potential to provide useful functions not possible by the individual units.

The goals of this project were threefold: (a) to develop and determine the effectiveness of a network that utilizes magnetically-connected and actuated modules; (b) to develop suitable self-reconfiguration algorithms; and (c) to extrapolate this macro scale network to future micro scale networks. Implementation of these goals was done in two major phases. Initially, sophisticated generic independent robot modules were designed and constructed to show that several modules could autonomously self-reconfigure in a laboratory network. The design of these modules was iteratively improved to arrive at a fully functional design. The second phase involved developing generic imbedded software for each module to permit the small network to move and self-reconfigure. The performance of the active magnetic actuation, passive magnetic latch robotic modules that were developed was observed to be comparable, but not necessarily superior to existing mechanically linked designs on the macro scale. The use of magnets vice mechanical actuators and latches, however, provides promise of being superior in micro scale networks of reconfigurable robots.

Keywords: Modular, Robots, Reconfigure, Magnets, Electromagnets

Acknowledgments

I would like to foremost thank my project advisors, Professor Kenneth Knowles and Associate Professor Brad Bishop for their dedication and support to my ideas. Thank you, also, to the following professors who lent a helping hand in my research: Professor Carl Wick and Associate Professor Deborah Mechtel. A special thanks to the Technical Support Division in Maury Hall: Ralph Wickland, Joe Bradshaw, Norm Tyson, and Sandra Erb. Thank you to the Machine Shop at the Naval Academy: George Burton, Art Goehring, Tom Price, Dale Boyer**.** Special thanks to Professor Joyce Shade for her dedication and support of the Trident Scholar Program. I also would like to thank the design teams at K&B Magnets and AWP Coils for their advice and expertise. Finally I would like to thank my friends for putting up with my stresses and lost social time. Thank you to my parents Marilyn and Albert Patterson for their love and support. And most of all, thanks to my fiancée Kimberly Wilson without whom I would have never accomplished what I have done.

Preface

Ordered clusters, or networks, of modular robots have the potential to be cheap, highly fault tolerant, and extremely adaptable, thus opening up the possibility of almost unbounded applications. This Trident Research is part of current worldwide research efforts to study modular robotics. It investigates novel approaches to modular network design and reconfiguration, adding to the existing body of knowledge of usable modular ideas. It is hoped that this research can find future applications in such diverse areas as medicine, space exploration, and search and rescue.

Table of Contents

List of Figures

List of Tables

1. Background

Reconfigurable modular robots are groups or networks of independent robots that work together to accomplish a common goal. Since the network is made of many robots, each individual robot is referred to as a module. The modules for this research are homogenous, meaning that each module is the same in every respect. The process of making and breaking the physical bonds between the modules as they reconfigure can be done via human interaction, or autonomously via self-reconfiguration. This research is focused on autonomous network reconfiguration.

In every modular robot network there is some mechanism for holding the modules together in the desired configuration. Most current research uses mechanical couples to link the modules together. As the overall size of the modules is diminished, however, the actuator and linkage sizes for mechanical bonding between the modules cannot be diminished proportionally (linear miniaturization). This problem provides the motivation for investigating static, non-moving bonding strategies, such as those which utilize electrostatic and magnetic technologies. The autonomous reconfigurable modules developed during the course of this project incorporate passive permanent magnets for latching, with temporary active electromagnetic flux modulation to effect the reconfigurations. This approach provides an energy efficient strategy that shows potential as a linear (or better) miniaturization technology.

2. Module Geometry and Material

The issues surrounding the shape and size of the module are by no means trivial. Though our original idea was to use a hexagonal shape (Figure 1), a symmetric six-point star shape

(Figure 2.) proved to be superior and has been used[1] throughout most of the research. The final layered gear module design was arrived at after experimenting with several different six-point star tip shapes. This final design is presented after a brief historical discussion of the geometric considerations.

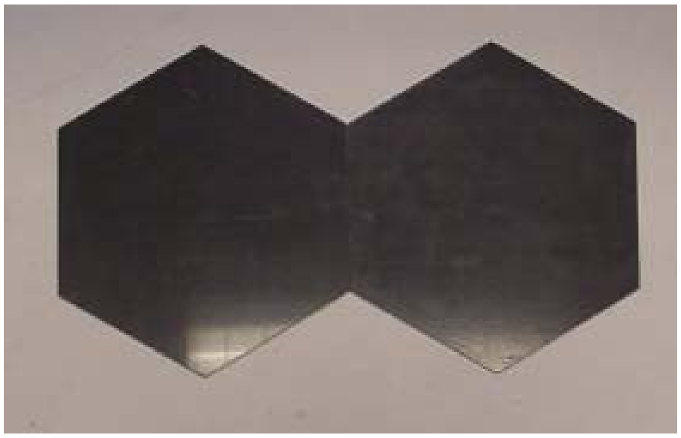


Figure 1
Two Hexagonal Modules side by side. These are made of 0.075” Plexiglas.

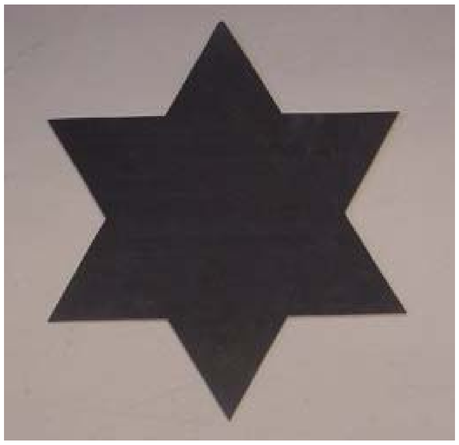


Figure 2
The first Six-Point Star Module that was made. The material is 0.075” Plexiglas.

A hexagonal shape initially appears to be ideal for two-dimensional reconfigurable modules. There are, however, four major advantages realized by using the six-point star shape in place of a hexagon. The first is that during rotational reconfiguration movements, the tips of the star create a stabilizing gearing effect. The second advantage is that the tip-crevice joint between stars provides a good docking point. The third advantage is that the star modules can better resist external shear forces with an “interlocking” geometry than the hexagonal shapes with their flat face connections. Finally, the angle of relative rotation per individual module movement of a six-point star is one-half that of a hexagon (60 degrees vice 120 degrees), thus permitting finer movement increments. The one major disadvantage of star module shapes is that they do not pack into as tight a group as the hexagons. There are triangular spaces at intersections of three or more star modules (Figure 3.), while there are no spaces with the hexagon modules.

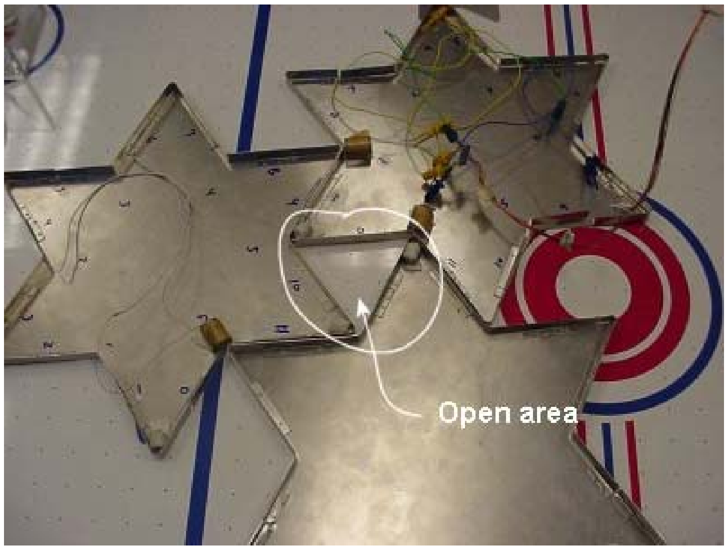


Figure 3

At the intersection of any three star shaped modules, a triangle of open area exists. This is a result of the Six-Point Star geometry. The most sides that allow for a tightly packed network (no open areas at any intersection) is six, and all interior angles must be obtuse.

The initial investigation of module geometry consisted of the design, fabrication, and testing of pairs of both hexagons and six-point stars (Figure 1, Figure 2). These first modules were made of Plexiglas, which as a material posed a unique problem. When used on an air table to impart the desired near-zero friction, these plastic modules and the table surface would build up a static attractive force over time that impeded movement. To alleviate this dilemma the next few designs were constructed with 0.037 inch aluminum (Figure 4). The only negative issue involved with the use of the thin metal designs was their tendency to bend slightly and create friction rub points when weight was applied, compared to the stiffer Plexiglas. Once the weight was suitably balanced and distributed, however, the metal modules floated fairly well on the air table and were much lighter than the Plexiglas ones.

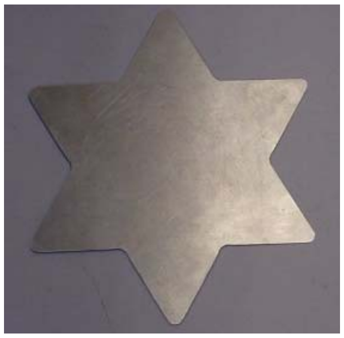


Figure 4

The first metal star made. This star is 10 inches from point to opposite point. It has rounded tips, yielding the advantage over its predecessors of not having the tendency of sticking in tight corners.

As the modules gained weight, there came a point where the low pressure air table operating surface could not "float' the modules sufficiently to eliminate friction. A solution to this problem was to use one-inch thick closed-cell foam floated in a shallow water bath to replace the metal on air table design. This water module consisted of a six-point star mounted atop a one-inch thick foam circle. In this manner, the edges in the water (the foam circles) were smooth and could spin easily, as opposed to having flat edges push through the water. As it turned out, we also discovered that the foam was more resistant to low stress deformation and much lighter than metal, and that it also could be used satisfactorily on the air table. An eight-inch[2] foam module could hold five magnets[3] and still maintain mobility, vice the larger 14-inch metal module's maximum permissible load (for air table use) of three magnets (Figure 5.).

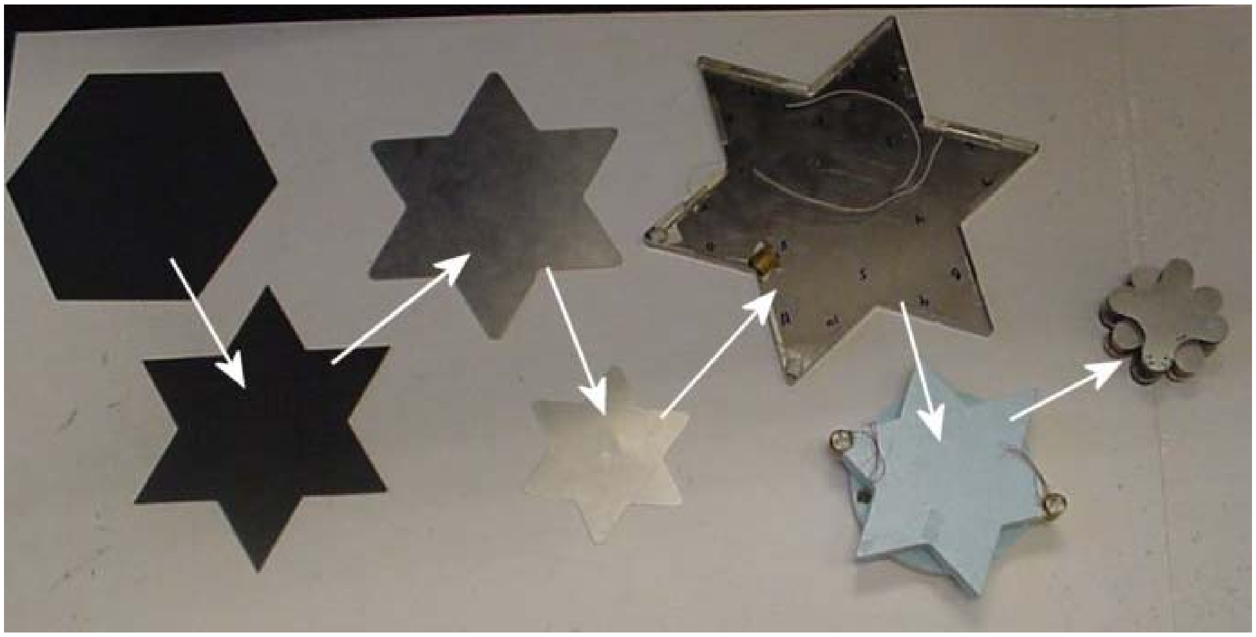


Figure 5
The evolution of the geometry and size of modules, starting from the upper left to the middle right.

Experience with the layering effect used in the star-on-circle modules, in combination with appropriate magnet placement (discussed in the next section), allowed us to see that if we used a three-layer module, we could place the electromagnets inside of permanent magnet fields. When de-energized, the iron cores of the electromagnets placed in this manner created much stronger permanent magnet attractions. By doing this and shrinking the size of the modules to four inches, using four electromagnets, four rare earth permanent magnet pairs, and rounding the star tips and crevices, we came up with our final module design, the layered-gear module (Figure 6, 7). All further module modifications and imbedded control program development was done with this module version.

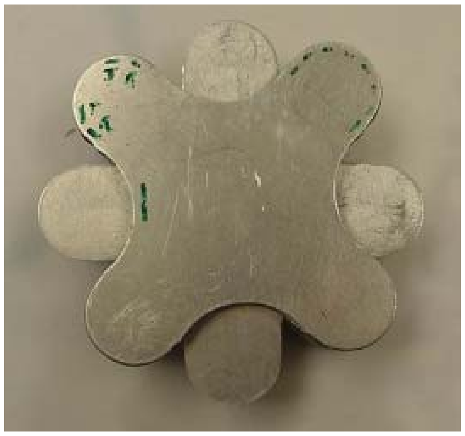


Figure 6
Top view of a prototype layered-gear module. Notice the middle layer is rotated 45° relative to the top and bottom layers.

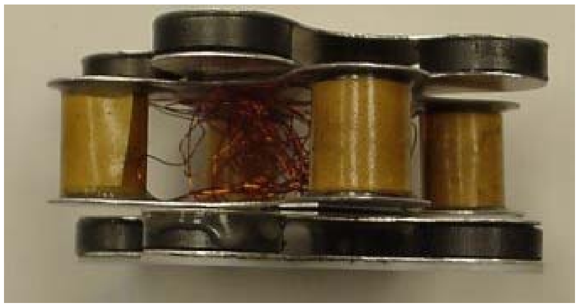


Figure 7
A side view at a prototype layered-gear module. The layering effect is clearly visible in this view.

3. Size and Magnet/Electromagnet Placement

When using the metal star with magnets on board, the minimum size seemed to be a function of the ability for the module to have enough surface area for the low-pressure air table to float the star. There is a point, however, at which a module can support all of its given weight, but it is too large to be effective in a network that will fit on the air table, or it is so large that the arc length of a movement[4] cannot be completed by the magnetic forces of the order of those we are using. Even with our best metal designs, only about three magnets could be added to the star without making it incapable of a single satisfactory movement.

Another issue surrounding size involves the potential for miniaturization of the modules. In the following discussion concerning module small-scale issues, the ability of a module to float friction-free in two dimensions on an air table or water pool will not be addressed[5]. Miniaturization issues end up pivoting around magnet arrangement: vertical (Figure 8) or horizontal (Figure 9).

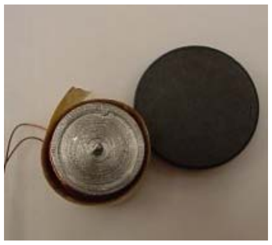


Figure 8
View showing the electromagnet and permanent magnet in vertical alignment. Flux lines are perpendicular to the plane of the page.

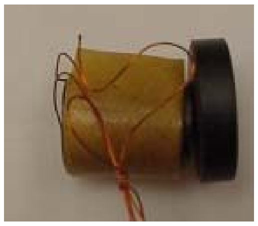


Figure 9
View showing the electromagnet and permanent magnet in horizontal alignment. Flux lines are parallel to the plane of the page.

In the case of two modules with horizontal magnet alignments interacting, there is a minimum module size at which the length of the arms of the star are so small that the attractive force between two adjacent permanent magnets exceeds the attraction force between one of the permanent magnets and its associated passive electromagnet. There is also a minimum size at which passive magnetic latching works, but with any temporary active electromagnetic repulsion used to invoke module repositioning, the permanent magnet attractions overwhelm the active repulsion forces, thus preventing motion. For the case of two modules with vertical magnet alignment, however, the problem of unwanted adjacent permanent magnet attractions is eliminated. Unfortunately, this configuration results in the passive latching field strengths on the sides of the magnets being less. Module repositioning movement is possible, but it takes more time to occur as there is less effective magnetic force.

As additional modules are added to the network, the problem of magnet placement and module size becomes more complex and difficult to balance. Most module array issues that arise occur at the level where there is an intersection of three modules. The following illustrates the remaining magnet and size issues for the multiple module case. In a triple intersection, all of the intersecting tips of the modules are nested inside other module crevices (between tips). In a horizontal magnet placement case, all magnetic poles are aligned correctly. For the case where the module size is large enough that no unwanted two-module attractions occur, the tip of the rotating module will pass just in front of a tip of the third module when one module tries to rotate out of a triple intersection. As it approaches this other tip, though, depending upon the size of modules, an attraction can occur between these tips if they are of opposite polarity or one is de-energized, pulling the modules together. Conversely, a large repulsion can occur between these tips if of the same polarity, thus preventing the moving module from rotating past the repelling tip and out of the intersection (Figure 10.). The blockage of movement due to repulsion is more obvious when the magnets are vertical.

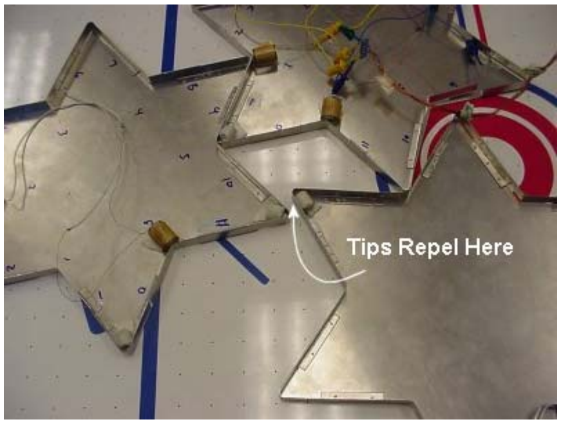


Figure 10

Repulsions and attractions can randomly occur at the intersection of more than two star modules with the permanent magnets on the tips of the star.

Our solution to this blocking problem was to reverse the positions of the magnets and electromagnets so that the electromagnets were on the tips and the permanent magnets in the crevices. This concept was first applied to the foam modules (Figure 11,12) and has been continued with the layered-gear module final design (Figure 13).

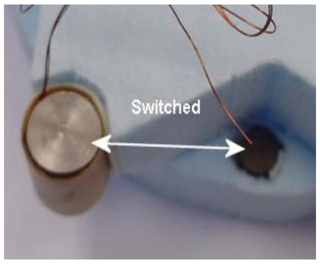


Figure 11
View of a water floating foam module with the magnets and electromagnet positions switched. The permanent magnet is in vertical alignment.

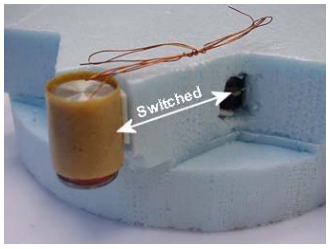


Figure 12
View of a water floating foam module with the magnets and electromagnet positions switched. The permanent magnet is in vertical alignment.

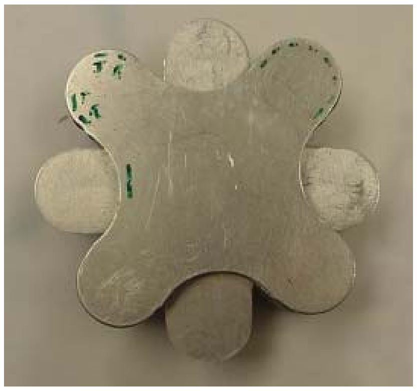


Figure 13

A top down view of the layered-gear module.

4. Explanation of Magnet Interaction

The final design, the layered-gear module (Figure 13), places the electromagnet inside the field that is created between a pair of one-inch rare earth magnets (Figure 14, 15). By using this configuration the de-energized electromagnets center themselves inside of the permanent magnet field. This eliminates most of the frictional interactions between module sides. With magnet placement, along with the size of the module, static (both permanent and temporary electromagnet) magnetic forces are able to effect a module rotational realignment relative to another module, thus eliminating the need to rely on inertial forces to continue and complete the rotation. The arc length of movement can be reduced and the required attraction and repulsion forces are much stronger.

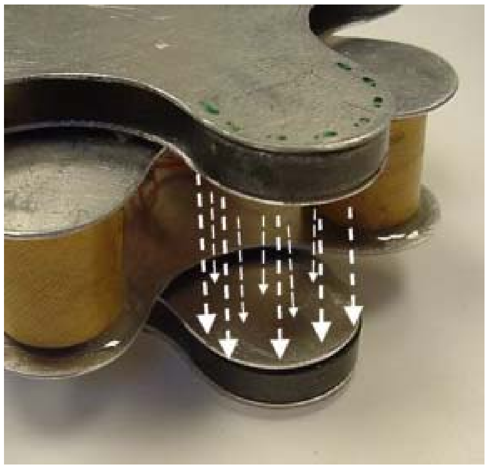


Figure 14

This view shows the layered-gear module's passive flux lines between its permanent magnet layers.
As shown in Figure3a.2, the electromagnet of another module would fit inside this field.

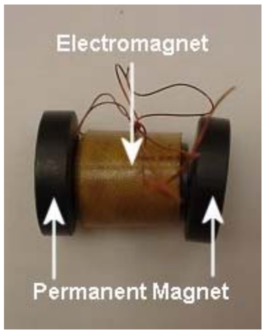


Figure 15
View showing how the electromagnets fit in-between the two permanent magnets.

When repulsion is excited in the electromagnet of module A by temporarily energizing it in a manner to create an opposite magnetic field polarity, it applies a force to the module causing the electromagnet to be pushed totally out from between the two permanent magnets of module B (Figure 16). At the point when the electromagnet of module A is totally out of line with the permanent magnet pair of module B, the next electromagnet on the adjacent tip of module A is almost in the field of its respective new permanent magnet pair of module B (Figure 17). The passive attraction between the new permanent magnet pair of module B and the adjacent tip electromagnet iron core of module B is often enough to complete the transfer. Temporarily energizing this adjacent tip electromagnet of module A in an attractive polarity can provide additional latching force, if necessary. This static magnetic force rotation sequence eliminates the need to rely on any momentum "swing" inertial force to affect the move. The direction of motion depends upon which side of the tip has its crevice latched to the other module. This crevice remains latched throughout the rotational move and suffices as a pivoting point. Because of the increased magnetic latching and rotating authority, there is no longer a requirement to operate in a near-zero friction environment provided by an air table or water trough. Teflon has been added to the bottom of the modules to reduce friction and aid in movement, but no air table is required for normal operation.

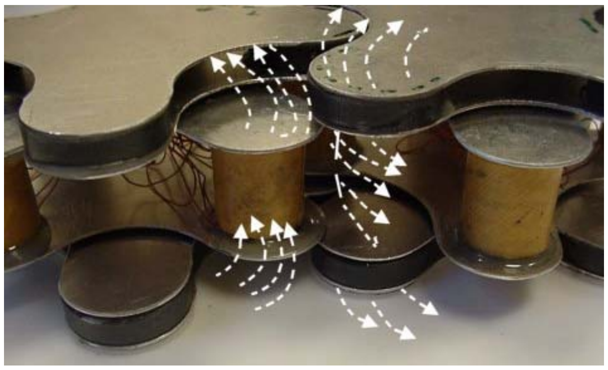


Figure 16

View showing how the flux lines of an excited electromagnet bend away from the flux of the permanent magnet field so that the module's arms move out of line. Note that the electromagnet must be active and must have an opposite polarity for this repulsion to occur.

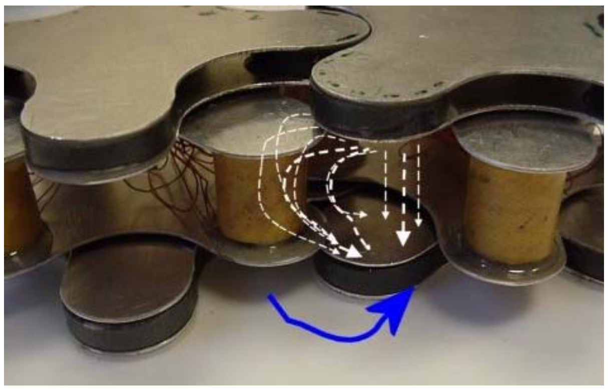


Figure 17

View showing how a passive permanent magnet field's flux will "grab" the ferromagnetic material of an electromagnet and pull it inline with its field. This happens because the permeability of air is much less than, in this case, steel. The electromagnet can also be excited in the proper attracting polarity to make this movement more expedient.

When repulsion occurs, the attracting pair of magnets (in the direction of rotation) acts as a rotation point by creating a moment using the magnetic interactions. Again, a large part of the frictional element is removed in this configuration since this point of rotation relies on the magnet forces to create a pivot and not the rubbing between sides.

5. Layered-Gear Module Design Issues

The major design issues that were identified and overcome were: a. the choice of type and size of permanent magnet; b. the length of the "arm" on the layers; c. the size, number of windings, core size, and wire gauge of the electromagnet; d. thickness and material of layer levels; and e. separation of layer levels

a. Type and Size of Permanent Magnet:

The permanent magnets used throughout the project were rare earth magnets. Ceramic magnets were considered, but their magnetic field strength to weight ratios are inferior to the rare earth magnets. While ceramic magnets are less expensive, purchasing rare earth magnets in bulk reduced their costs almost 90%. The overriding permanent magnet characteristic was adequate magnetic field strength in a small size, thus dictating the use of the rare earth magnets. Original air hockey table designs (hexagon and six point star) used a 0.25 inch thick, 0.25 inch diameter rare earth magnet. The final design uses a 0.25 inch thick, 1.0 inch diameter rare earth magnet which is nickel plated. This size was chosen since it was the smallest size that could be used and still have the Rabbit microprocessor fit horizontally on the top of the module (Figure 18). This permitted the magnets of two interlocking modules to be virtually side-by-side, which provided the tightest packing of the magnets that still allowed for the processor to fit. This rare earth magnet was the constant in the rest of the layered gear module design topics.

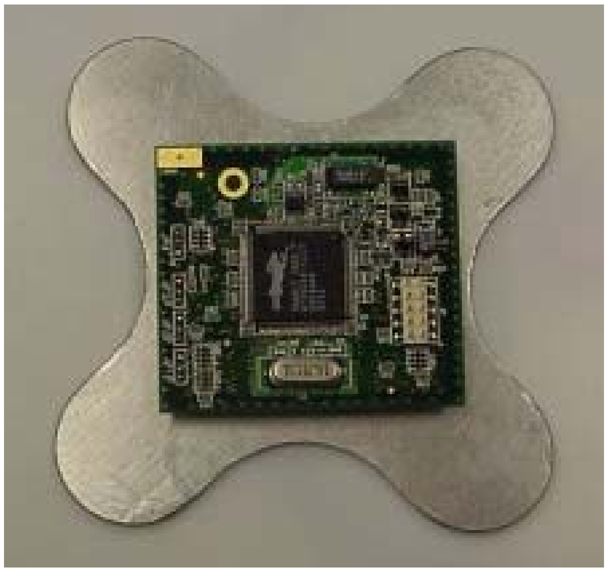


Figure 18

This figure shows that the chosen layer size is sufficient to permit mounting of the microprocessor board on the top layer.

<u>b. Length of the "Arm" on the Layers:</u>

The original module size was chosen to be 3.5 inches from tip-to-tip. This was the smallest feasible size that permitted using the one-inch permanent magnets. Since the magnetic fields of each module are aligned vertically, a repulsion force is created between the vertical stacks of two adjacent modules. It was learned that a field can be directionally decreased, but there is some resultant decrease in all directions if this is done. The effective weakening of the field is done by adding ferromagnetic material to the edges of the permanent magnet around the circumference of the disk. This short circuits the flux, and thus highly limits the range of the flux of the permanent magnet that comes out of the sides. Though it also has an effect on the non-shielded sides, the ratio between the strength of the side to that of the face is greatly increased.[6] The exact analytical equation for how much the flux out of the face of the magnet is affected by this shielding is not yet known. In a case where the circumference and one face are shielded, however, it was found that the uncovered face's flux falls off much more rapidly, proportional to R^{-8} vice the normal R^{-2}.[7] Intuitively, by just surrounding the circumference with galvanized steel, the flux out of the ends can be expected to fall off somewhere between R^{-2} and R^{-8}.

Since the original production of the layered-gear module, a semi-circle of 0.075 inch galvanized steel has been added to the outboard sides of every magnet (Figure 19.). The arm length (Figure 20.) was also extended 0.375 inches per side, or 0.75 inch on the tip-to-tip measurement. The final module dimensions were arrived at experimentally to assure proper docking, undocking, and pivot point rotation (Figure 21.) of a module without ejecting an adjacent docked module.

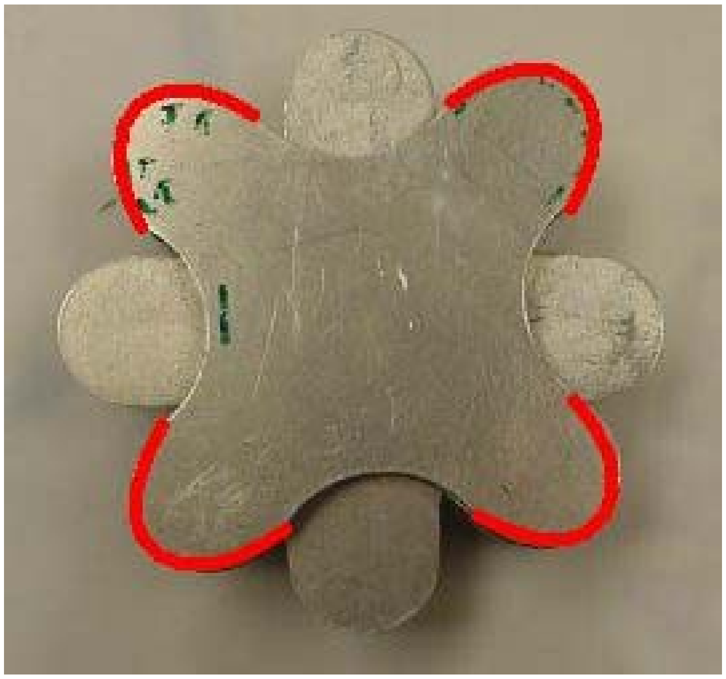


Figure 19

This figure illustrates where the galvanized steel semi-circles are added to the sides of the permanent magnets. This addition essentially short circuits the magnetic field and therefore cuts down on the strength of the field, most noticeably in the direction that the ferromagnetic material is applied.

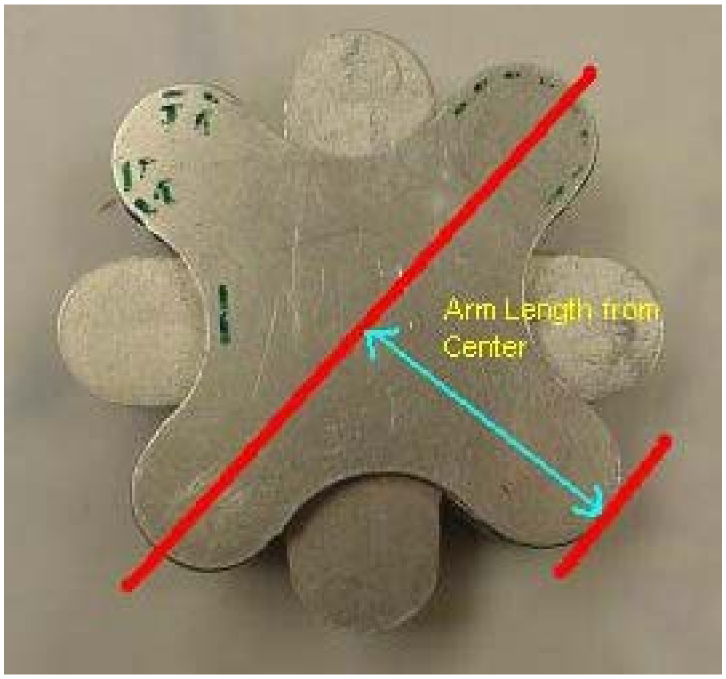


Figure 20

Arm length is measured from the centerline just like a radius.

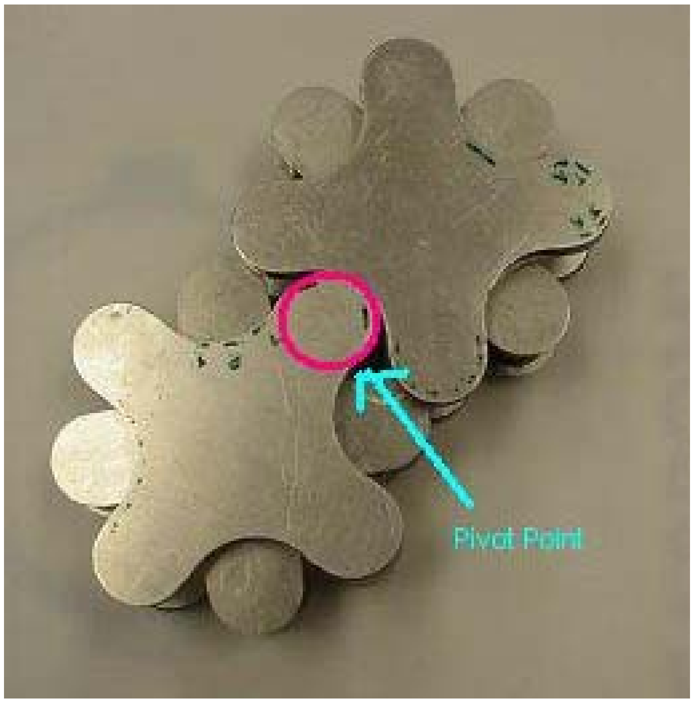


Figure 21

The pivot point is the point where, during a move, the electromagnet of one module stays in the permanent magnet field of the other module. This is an essentially stationary point during rotation.

c. The Size, Number of Windings, Core Size, and Wire Gauge of the Electromagnet:

The design of the electromagnet goes hand in hand with the arm length. The two were experimentally determined concurrently. The size of the electromagnet's outer diameter (OD) was desired to be one inch, however to order an electromagnet with a desirable length, the availability of stock spools forced an OD of 0.875 inches.[8] Final experimentation with this OD yielded a 28-gauge wire with 625 windings around a steel core of 0.13 inch diameter and a total length of 0.58 inches.

The number of windings and wire gauge acted together in the relationship of amp-turns. The amps are a function of resistance of the wire, which the gauge and length of wire determine. Since the number of turns determine the length of wire, it was not possible to just add as many windings as desired without affecting the amps via wire resistance. Therefore, the amp-turns are a balance of number of windings and gauge size. We were limited to an OD of 0.875 inch, thus there were a maximum number of windings (depending on gauge) that could fit on that particular spool.

The inner diameter (ID) and length of the spool dictated the size of the core of metal that was placed in the electromagnet. The core had to be large enough to have a good passive magnetic connection between modules, but be small enough that the available amp-turns would excite a large enough force to overcome and repel the rare earth magnetic field. All of these variables were experimented in conjunction with the layered-gear arm length and eventually led to a feasible module design.

d. Thickness and Material of Layer Levels:

Aluminum metal comprises the layers of the layered-gear module that hold the permanent and electromagnets. To reduce friction, the final modules have a Teflon tape layer that is approximately 0.019 inches thick applied to the bottom layer. In addition, this same tape was also applied to the pivoting parts (Figure 21) of the module's arms (Figure 20) to reduce friction there. The original prototype's layers were made of 0.037 inch aluminum. It was discovered, however, that because of the strength of the rare earth magnets and the relative flexibility of 0.037 inch aluminum, the layers deformed under the stress of the upper and lower levels attracting each other. This bending caused the epoxy that bonded the magnets to the aluminum to crack and eventually the magnets would break off. Sandwiching the magnets between two layers of metal significantly reduced the bending strain. The upper and lower levels of rare earth magnets still pulled together enough to slightly close the gap between the layers, however, physically preventing an electromagnet from entering the space (Figure 22). The thickness of the metal could not be increased too much, however, since the magnet strength falls off at R^{-2} and much of the previous design was done assuming a certain separation. Increasing the lower level upper metal layer to 0.075 inch sufficiently reduced the bending strain due to the magnetic attraction force to an acceptable amount. The bending of the sandwiched lower level is not so severe as to close the gap.

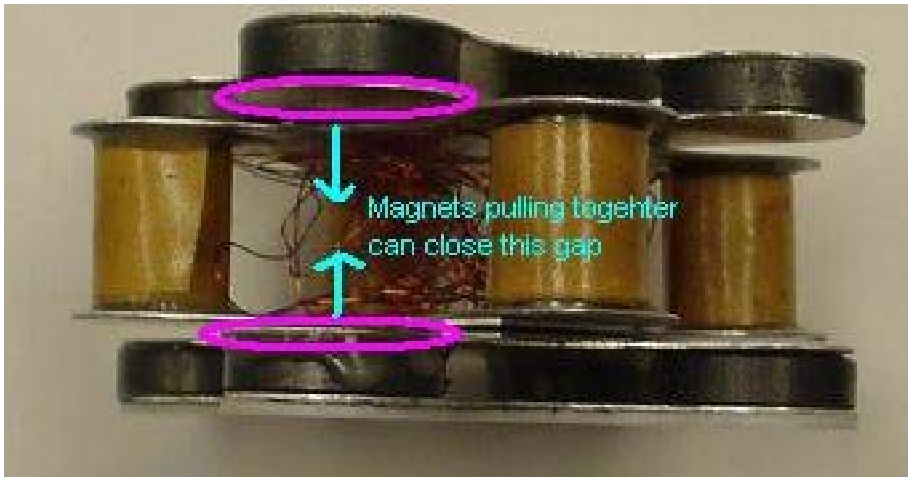


Figure 22

This view shows where the attractive force of two permanent magnets can close the gap between them.

<u>e. Separation of Layer Levels:</u>

This design item played hand in hand with the thickness of the metal. It is again noted that the separation between the layers cannot increase too greatly due to the rapid fall off of magnetic fields. There has to be some separation between the levels for the electromagnet to fit into the gap between the upper and lower permanent magnet levels in a self-reconfiguration. Therefore, spacers were added on the top and bottom. Spacers (0.075 inch) are used in the final design to prevent any upward bending of the bottom layer from closing the gap. This 0.125 inch "play room" allows the module to freely rotate even if the layers of the module are not completely flat or aligned.

6. Dual Star Movement

The original version of the movement program, designed in Dynamic C™ for operation on the Rabbit 3100™ (Figure 23), was fully programmed for the movement of two six-point star modules.

Figure 23

The Rabbit 3100™ has 512 Kb RAM and 512 Kb Flash ROM. It has six serial ports and 54 I/O ports. The Rabbit has the capability of pulse width modulation and well as full I/O register control.

The program was successful in the rotation of two modules about each other from any side to any side (Figure 24,25,26). It also succeeded in conducting parallel communication between two modules.[9]

Figure 24

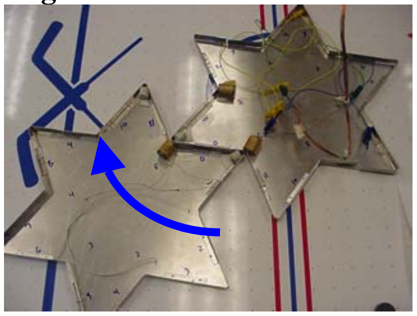


Figure 25

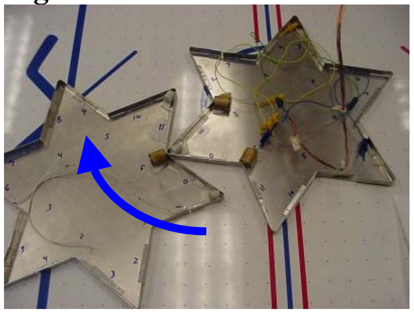
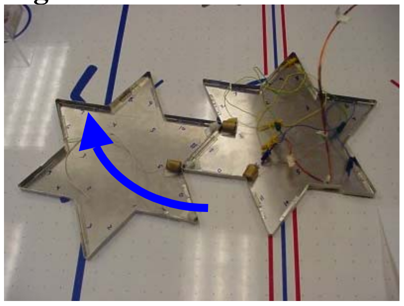


Figure 26



Three views showing a successful move of two six point star modules on an air hockey table using the original Dynamic C Program. A complete right (counterclockwise) move is from Figure 24 to Figure 26. In Figure 24 the right module repels the left module by exciting its electromagnets opposite to the permanent magnets to which it was attracted. In Figure 25 the right module de-energizes all electromagnets while the left module attracts the right module tip to its crevice to create a rotation point. In Figure 26 the right module crevice attracts the left module tip so that the two modules dock properly. Once the modules have fully docked, all electromagnets are turned off.

For a fully documented version of the control program see Appendix A.

The dual six point stars only moved in right (counterclockwise) rotation[10]. A move is shown in the sequence from Figure 24 to Figure 26. In Figure 24 the right module repels the left module by exciting its electromagnets opposite to the permanent magnets to which it was attracted. In Figure 25 the right module de-energizes all electromagnets while the left module attracts the right module tip to its crevice to create a rotation point. In Figure 26 the right module crevice attracts the left module tip so that the two modules dock properly. Once the modules have fully docked, all electromagnets are turned off.

To conserve power, the repelling electromagnets are only pulsed for about 0.3 seconds and then turned off before the attracting magnet is turned on. The attracting magnets stay on until the move fully occurs, i.e. the modules dock. Once the move is complete the attracting magnet shuts off.

Since all commands to a module are right moves, the moving module determines which modules need to energize which electromagnets and the polarization of those magnets. If the starting side number[11] is odd and receives a move right command, it will inform the other module to become the master (the one that uses two electromagnets in the move) and makes itself the slave (the module that uses one electromagnet). The master module always does the repel-attract sequence as described above. The slave always creates an attraction at the pivot point so the modules do not become separated during the move.

In the communication between two adjacent modules only one parallel bit is used. This bit tells the other module that the transmitting module has received a move right command. The receiving module then determines if it is a master or slave by detecting if its connections are odd or even.[12]

7. Movement Program (final version)

The final program implemented on the layered-gear module design was derived from the original six-point star program. The underlying concepts are the same with a few exceptions: a) the final program will let a module move clockwise or counterclockwise itself instead of relying on the relative movement of the another to accomplish a clockwise movement; b) the number of active sides in the final design is eight and the number of electromagnets controlled is four; c) communications are done via an IR pair on each of the eight sides; d) sensor information is actually gathered rather than simulated; and e) the processor can receive commands via the Serial C port, vice the parallel port inputs of the original program.

The fully commented program listing is attached as Appendix B. A brief explanation of the program is given below. It should be noted that the program of Appendix B is functional, but may not fully be debugged for all cases.

Once the program receives a command from the reconfiguration algorithm it flags a move variable. This will cause the program to look at the active side matrix. The active side matrix is a matrix that contains the sides which have another module's electromagnet between its permanent magnets or has its electromagnet between another module's permanent magnets (Figure 27). The program compares the movement command (clockwise or counterclockwise) to the values of the active matrix. It then determines which electromagnet to excite and which direction the phase should be (see section 9). It will send commands from its communications port to other modules that need to be actively involved in the move to engage their electromagnets and what polarities they need to set. Once a move is complete, the program will look for the correct active side matrix to occur and then disengage its electromagnets.

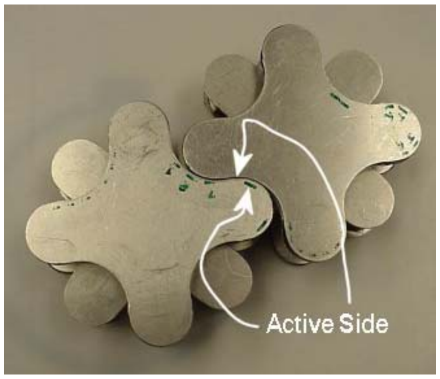


Figure 27

View showing which side of each module is defined as the active side. The active side is always defined along a long side, i.e. not along a curved surface.

8. Serial Communications

The final module design uses serial communication between modules. The communications occur through one serial port (port B) and use a direction bit to determine what side (Figure 28) on which to transmit the serial data.[13] This occurs through the use of an AND gate that uses the serial data and direction bit to direct communication flow. The transmitter/receiver channels utilize fairly directional IR LEDs that are spaced throughout the eight sides (Figure 28). Due to the geometry of the way the modules dock, there are two receivers for every one transmitter.

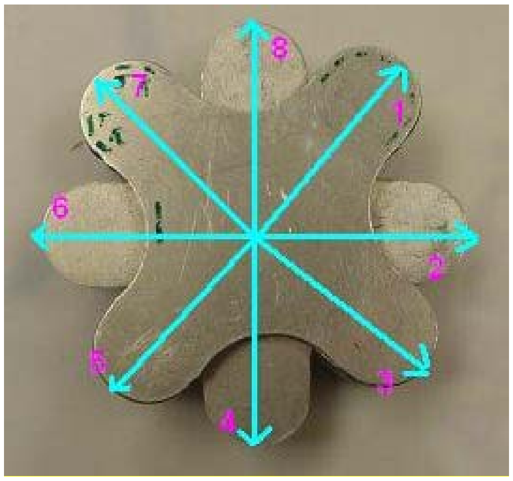


Figure 28

This markup view shows the available number of sides (eight) to which the Rabbit can individually control communications.

The receiver implementation is slightly more complicated than the transmitter. Because only one serial port is used, the module is unable to determine from which side data is coming, based on the data stream alone. To solve this problem the module uses a Generic Array Logic (GAL) programmable logic chip (see Appendix C for code). Each side has a signal conditioning circuit that feeds a serial data stream into one of eight inputs of the GAL that correspond to one of the eight sides of the module. Once communication occurs from one of the eight inputs to the GAL, it puts a high bit on one of eight corresponding outputs. This high logic output will not fluctuate with the data stream until a reset bit comes from the Rabbit. This output bit is read into the Rabbit, thus allowing the module to know what side the communication came from since each side has its own respective output pin. This scheme prevents other modules from talking to a module that is already using its communications. Serial communication that is streaming from any side comes out the data output pin that is fed into the Rabbit's serial B receiving port. To eliminate extraneous noise, only sides that are active permit communication[14].

9. Sensors

Originally the layered-gear module was designed using eight photodiodes as sensors to trip when another module joined the sensor's corresponding side. In practice it was found that these diodes were too fickle with regard to different ambient lighting conditions and the signal processing circuit took a great deal of physical room. Miniature opto-reflective proximity sensors were found to eliminate these issues. They were used for the final versions of the modules.

These sensors require very little signal processing. They use one NOT gate to convert the transistor current from the sensor into appropriate voltage levels. These sensors have worked flawlessly in all room lighting conditions, thus permitting the Rabbit to create its required active side matrix. This matrix contains the only world information gathered by the Rabbit microprocessors on each module, so these sensors must be virtually infallible.

10. Electromagnetic Activation and Power

The electromagnets are excited through a appropriate driver circuit. The circuit uses an h-bridge power amplifier (LM298) to control on/off and direction of the electromagnet's field (Figure 29). This amplifier is controlled using a phase bit and an enable bit that are part of the programming. Since the electromagnets are only pulsed for about 0.15 seconds, there is no problem with excessive heat buildup from momentarily large current flows.

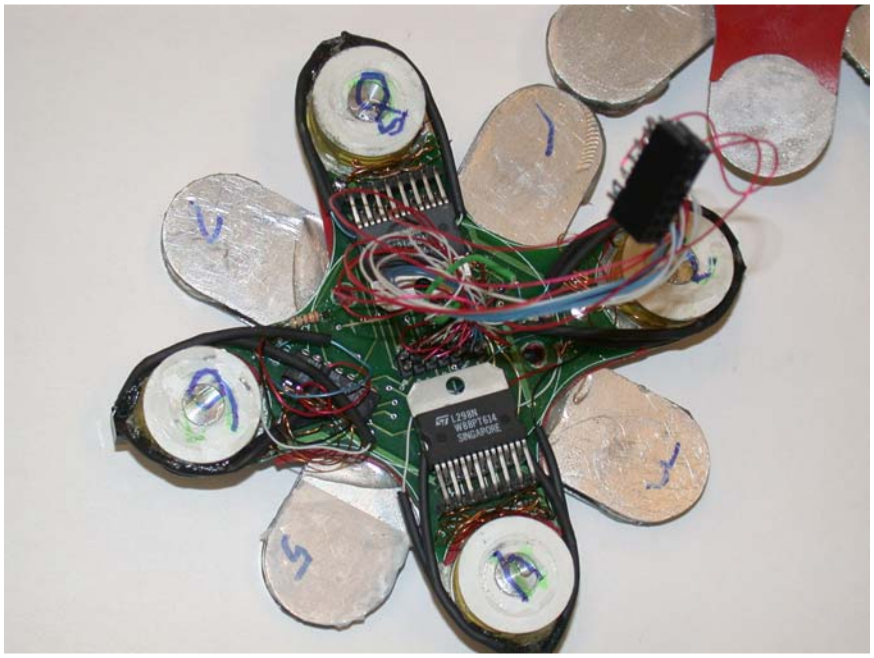


Figure 29

This board contains two h-bridge power amplifiers that use the phase and enable bits to determine on/off and polarity of their respective electromagnets.

The power amplifier is capable of drawing its high current from Polapulse Batteries that can fit atop the module (Figure 30). This is a high-density 6-volt battery that can deliver 7.6 amp hours of energy at a one amp drain. The power amplifier battery is only connected to the h-bridge for electromagnet control and does not drive any other part of the robot.

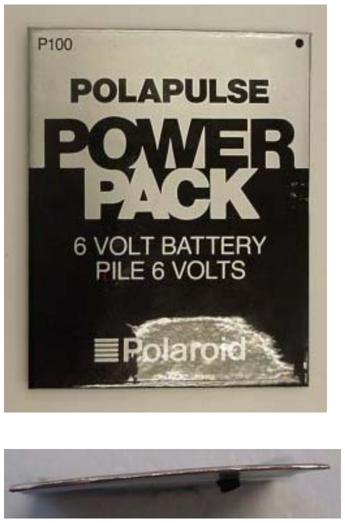


Figure 30

The Polapulse Battery can deliver 7.6 amp/hr at 1 amp to 5.5V. It is highly compact as can be seen from the bottom view. Its size allows it to fit vertically on one layered-gear module.

11. Module Power and Power Consumption

As stated in the Electromagnetic Activation and Power section, the Polapulse Batteries can be used as the power source of the modules. Three batteries are required to supply 18 volts which drives the electromagnets at about 2 amps. A separate Polapulse, via regulation, powers the Rabbit microprocessor and other logic. A 6 V high amp/hour battery is used because it is extremely lightweight vice something like a common 9 V battery.

Success has been realized on the air table driving the module with as little as one watt-sec per move (5.5V @ 0.8 A for 0.25sec). The layered-gear module, on a dry surface, draws around 4.86 watt-sec per move (18V @ 1.8A for 0.15sec).

12. Circuit Board Design

The circuit boards on the final module where designed and printed. To maximize usable circuit board area without impeding module movement, the circuitry is divided into four circuit board layers. They are shaped to match the aluminum layers. The top three boards connect via headers and are atop the top layer of permanent magnets. The fourth layer is inside the middle layer of electromagnets and connected to the third layer via wires through the middle of the top permanent magnet layer. Table 1 shows what each layer does and Figures 31,32,33, and 34 show the layers. Figure 35 shows a side view of the assembled module.

TABLE 1: LAYER FUNCTIONS

Layer (top down)	Function
1	Rabbit Processor, buffering, and power regulation
2	Communications
3	Sensor Processing
4	Power Drivers

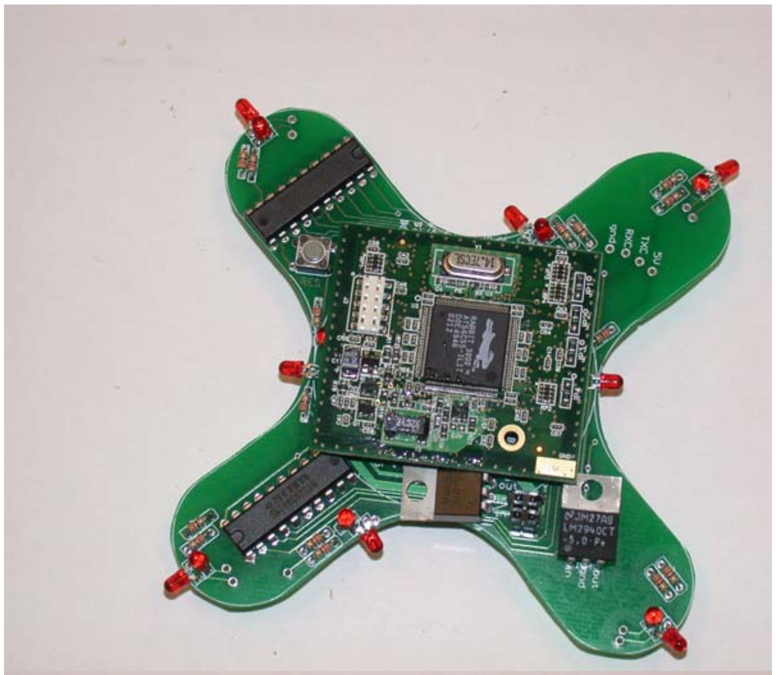


Figure 31

Top Layer: Contains the Rabbit 3100 in the center with buffer latches surrounding it. The red LEDs signal the sensor inputs and the enable bits going to the electromagnets.

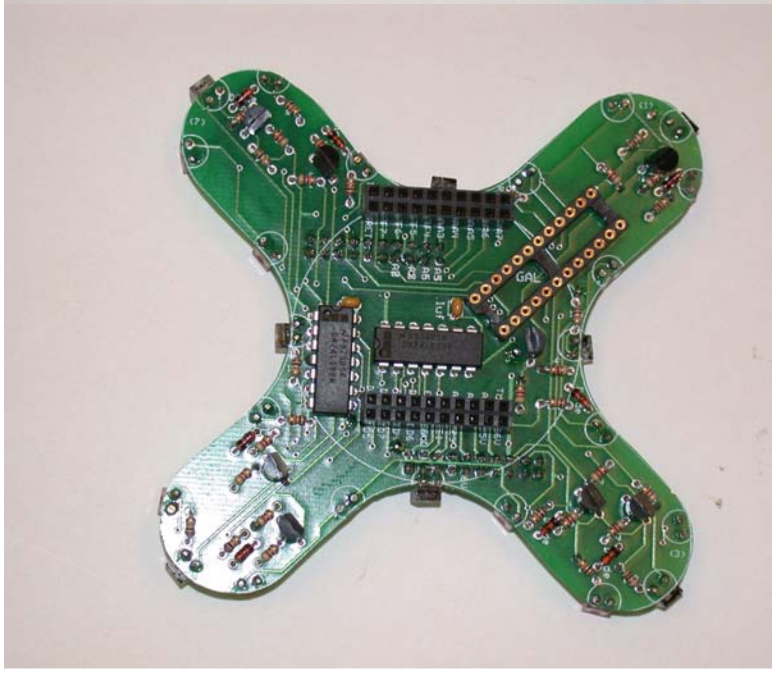


Figure 32

Second Layer: This layer communicates via IR to other modules (the IR's are underneath). The missing component is the GAL, which does much of the receiving signal processing.

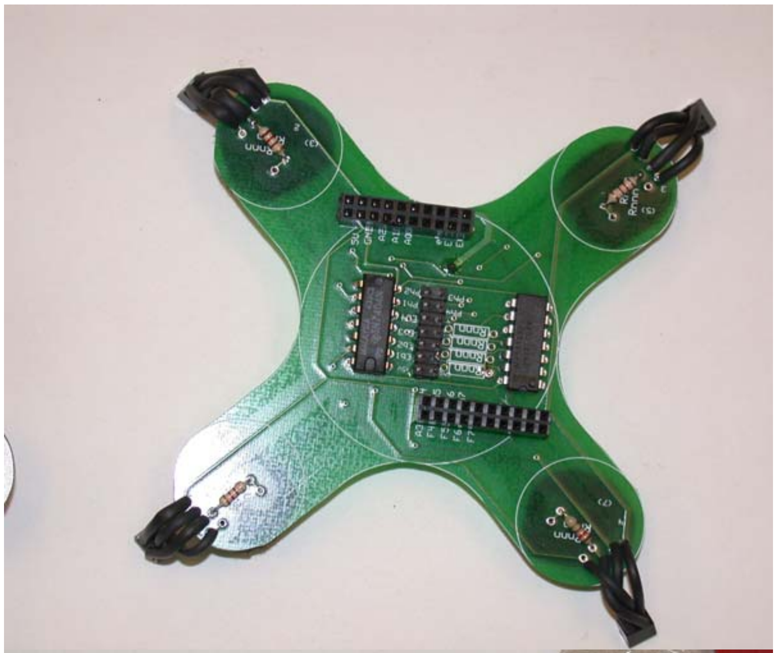


Figure 33

Third Layer: This layer does the sensor processing. It uses a NOR gate for each of the opto-reflective sensors (four seen on this layer)

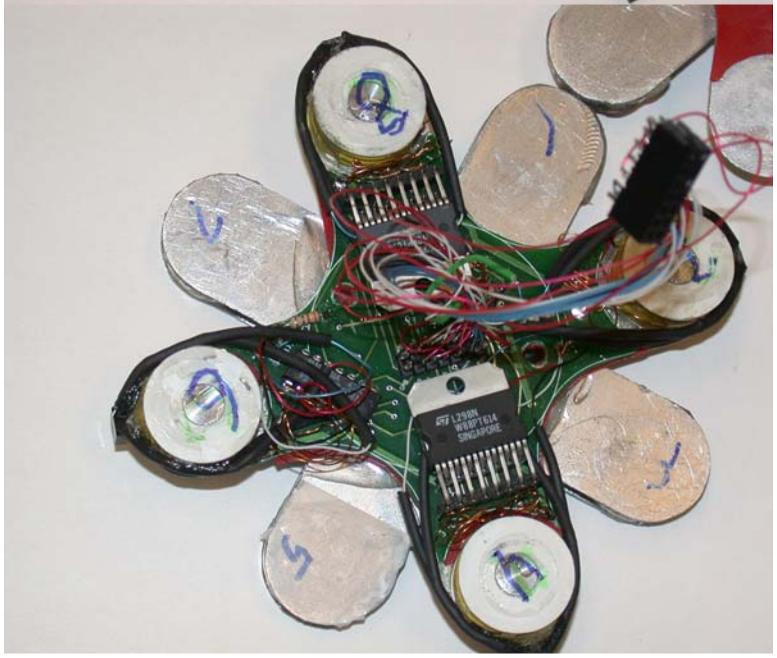


Figure 34

Fourth Layer: This layer contains the remaining four opto-reflective sensors. There are two LM298 boards and a cable that connects to the third layer.

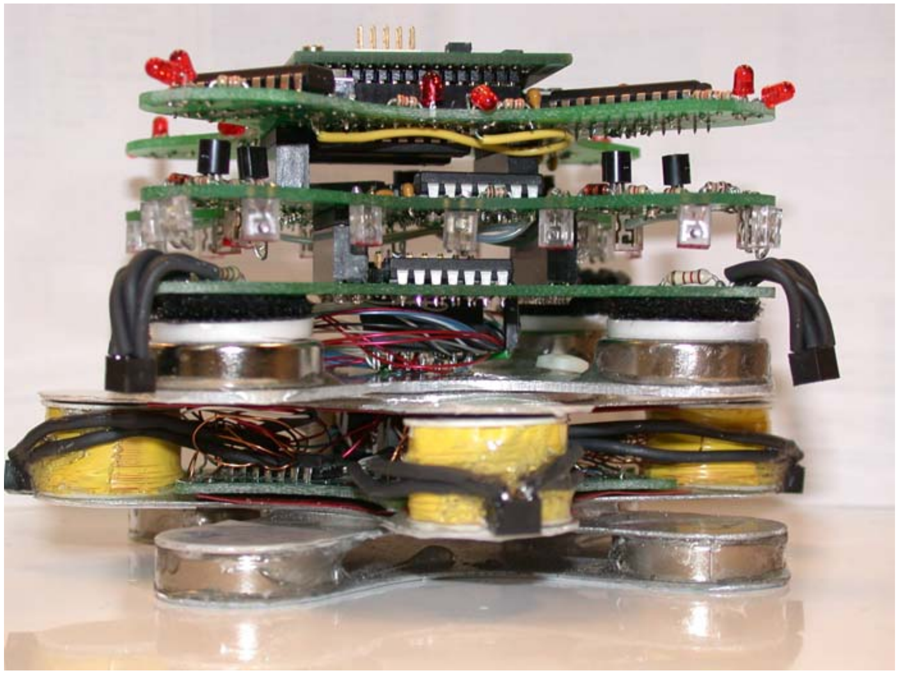


Figure 35

Side view of a layered-gear module. Top layer: Processing; Second layer: Communications; Third layer: Sensor signal processing; Fourth layer (inside of the yellow electromagnets): Power control

13. Current Ideas on Self-Reconfiguration

When considering self-reconfiguration, the main goal is to achieve successful reconfiguration with the least amount of local knowledge. This goal leads one to examine how humans work. What do we know about the world around us when we participate in a group that accomplishes a task with no single person directing the task? We generally move for the betterment of the group. In a calm group, where we do not push or shove, a reconfiguration of the group is usually a smooth process that arrives at a general form and then refines it.

An example of a human group reconfiguration is when hordes of people are outside a building and must form into a line to get through the door into the building. Normally the line is pretty much formed as an individual reaches the door, so it is not a function of saying "We must reconfigure when approaching the door." Humans tend to get into a "line" formation before getting close to the door. There is no person with a megaphone telling them to move north, south, east, or west. They move because they "feel" and observe what the rest of the people around them are doing with respect to the building whose door is the target. The person's height of eye does not usually tower over everyone to see where the group is moving. Therefore, the only thing they know is that they see a building and there are people around them. Eventually the group does form a straight, semi-single file, line and achieve its goal of moving through the door.

It is not the objective of this research to develop something that will move through a door, but rather just to get into the line. This latter objective seeks the answer to the question, "Why did the person move this way or that way? What did they know about the world around them that helped make those decisions, and did they have more information than they needed to get into this line?" What we know: 1) There is a building and about where the door is relative to us. 2) If we could only see in eight directions - the points of an octagon around us - there are up to eight (in the case we will first consider) people around us.

Now that we know there is a building and a door; what does that mean to us and how do we translate that into a mathematical representation? We can see the building because it stands high so we know its direction, or in mathematics, a unit vector that points toward the building. We know approximately its distance or, in mathematics, the length of a vector. So now we have a vector that points toward one point in space with an approximate length.

We also know that there are people around us. People have given us what we call active sides. We can have a matrix of people that are next to our eight sides. There is either a person located there or not. Who the person is does not matter; in communications this means that we will not assign unique numbers to each module. The non-assignment of numbers is a big deal when trying to relate this reconfiguration scheme to a smaller scale with millions of modules. Number assignment and timed communications will overload computation.

Looking back at what we now know in terms of mathematics: 1) We know a vector that points in an approximate direction and has an approximate length. 2) We have a matrix of on/off data concerning our eight sides. The modules can locally obtain this matrix data, but we must give it a reference for what we call the module goal vector.

14. Future Reconfiguration Work

There are two options being considered for establishing the goal point. One is to randomly make one module the goal point and communicate the information about where that module is outward through the formation. The other option is to place some sort of radio frequency beacon or optical reference point around the modules as the goal point. In this case the modules would need to determine their orientation to the goal point.

The way the reconfiguration would work is that the goal formation of the modules would be communicated in terms of a series of vectors. These vectors would represent the distance and direction to the location where that a module should be in the goal formation. The vectors that we call the universal goal vectors extend from the goal point and end at every desired module location (Figure 36).

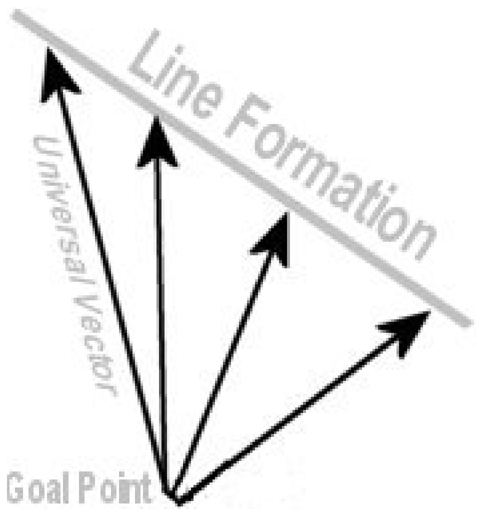


Figure 36

This shows how universal vectors are defined from a goal point out to points which define a formation. A goal point can be either another module or some object in space that can communicate to the modules its location, e.g. a radio beacon.

If a formation has thickness to it (i.e. more than one module along the same vector direction) the universal vector will add sub-universal vectors from its end point to the start point where the next module should be. The number of sub-universal vectors can be large (each is numbered according to the number of vectors it is away from it's respective universal vector). For instance, the first subvector after the universal is labeled one. For simplification, a point where a subvector or universal vector ends can also be the start of a tree subvector. The start of a tree subvector is where more subvectors can form. What results from all of these vectors and positions is a formation defined by what is known to graph theory as a tree (Figure 37)[15] [16]

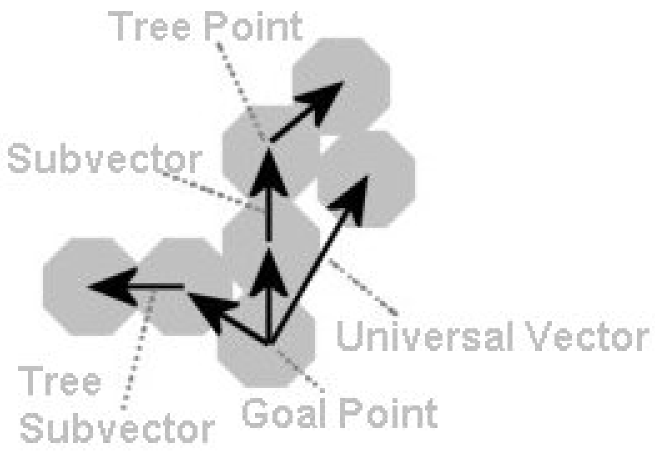


Figure 37

This Figure shows all the different types of vectors in a tree graph that defines a formation. All universal vectors originate from the goal point. A tree point allows tree subvectors to grow off either a universal subvector or a universal vector. All formations are defined with tree graphs that, by definition, do not have cycles in them.

Since this reconfiguration can now be seen in terms of graph theory, there are many well-established theorems that can be used to gain facts about the goal network in relation to one module's position. For instance, Kruskal's Algorithm will allow us to make the least number of moves to get from one position to another. There are numerous graph theory analysis tools that are being studied for possible application to this reconfiguration scheme.[17]

15. Nanoscale – Feasibility and Goals

One of the major goals of this project was to make it applicable in the future to the nanoscale. The problem of power consumption is a major issue that must be resolved before onboard power on micro and nanoscale robots can be realized. For the motion actuation (magnets), the weight to strength ratio and, therefore, theoretical power consumption, decreases drastically as the size and mass of the modules decrease. This reduction is due to the fact that magnetic fields decrease proportional to R^{-2}. Though downsizing theoretically increases the strength to weight ratio, the actual interactions between separate magnetic fields become a major issue not easily analyzed. Due to the subtleties of quantum and other effects, this problem becomes even more acute as miniaturized modules approach nano dimensions.[18]

Network communication complexity has been limited by eliminating the need to label modules, use pseudo random numbers to assign communication time slots, or otherwise assign an identification for communications. Every module is considered to be the same. It does not matter which one is which, only that a goal formation is either achieved or not. The communications scheme that has been developed has accomplished this goal.

16. Future Research

Future research should initially address the full implementation of the reconfiguration algorithm. Simulations will need to be developed once the dynamics of large networks of these new modules are known and can be modeled. Once faithful simulations are available, reconfiguration algorithms can be refined and studied, and new algorithms implemented.

Module design can be further optimized. More time and funds should be invested into the current prototype to fully realize its potential. Certain design parameters that the current module uses can be changed and studied. It is hoped that a smaller version of this current module design can be constructed in the future. Even if full onboard realization of all control and power is not possible in miniaturized versions, future research still could prove more decisively that magnetic coupling and actuation is a preferred choice for future micro scale modular robots.

Endnotes

[1] The hexagon, the shape proposed in the original design paper, was built along with the first six-point prototype but was never implemented with magnets. This design was discarded and the six-point star, which had been considered over the summer, became the working shape.

[2] Module dimension refers to the diameter of the smallest circle that could be used to fabricate the module.

[3] This refers to both permanent and electromagnets, however the original electromagnets were orders of magnitude greater in mass than the permanent magnets. Most discussions about weight and mass can be referenced as almost entirely to numbers of electromagnets.

[4] The distance it takes for a module to "swing" from one connection to the next. Original design concepts relied heavily on the dynamic inertia of a module to swing this gap.

[5] Floating, though it refers to an air table here, is encompassing all frictionless environments.

[6] K&B Magnets, Phone Conversation, 30 Oct 02.

[7] K&B Magnets, Phone Conversation, 30 Oct 02.

[8] Non-stock spools can be special ordered, but lead time and cost are too great

[9] This communication was hardwired. It did not use IR.

[10] In the original program all movements were thought of as to the right. To get a module to move left is accomplished by telling the other module to move right. Everything is relative.

[11] Numbered by convention from 0 to 11. Numbers of the sides to the left of the tips are odd and the numbers to the right are even.

[12] For two modules, odd and even active sides occur in pairs. The two modules could not be attached and have one side with an odd and the other with an even.

[13] Rabbit Semiconuctor. "RabbitCore™ RCM3100 User's Manual"
http://www.Rabbitsemiconductor.com/documentation/docs/manuals/RCM3100/UsersManual/, 02 SEP 02.

3[14] This is accomplished in programming, though originally designed in gate logic.

[15] Harris, John, Jeffery L. Mirst, and Michael J. Mossinghoff, "Combinatorics and Graph Theory." Springer: New York, 2000.

[16] The method of reconfiguration was developed before graph theory was applied. The reconfiguration scheme that we have developed easily fits into the context of graph theory. Therefore, during development, we were not trying to fit inside the box of graph theory. Rather, it was seen afterward that graph theory's box could fit over our method.

[17] Harris, John, Jeffery L. Mirst, and Michael J. Mossinghoff, *Combinatorics and Graph Theory.* New York: Springer, 2000.

[18] Dresselhaus, M.S., G. Dresellhaus, and R. Saito, *Nanotechnology*. ed. Gregory Timp. New York: Springer-Verlag New York, Inc., 1999.

Bibliography

Dresselhaus, M.S., G. Dresellhaus, and R. Saito, *Nanotechnology*. ed. Gregory Timp. New York: Springer-Verlag New York, Inc., 1999.

Harris, John, Jeffery L. Mirst, and Michael J. Mossinghoff, *Combinatorics and Graph Theory.* New York: Springer, 2000.

K&B Magnets, Phone Conversation, 30 Oct 02.

Rabbit Semiconuctor. “RabbitCore™ RCM3100 User's Manual“ http://www.Rabbitsemiconductor.com/documentation/docs/manuals/RCM3100/UsersManual/, 02 SEP 02.

Yim, Mark. “Modular Robots” heep://www.parc.Xerox.com/spl/projects/modrobots/, 12 Nov 02.

Yim, Mark. “Re: Research.” E-mail. 5 Dec 01.

Appendix A

Original Dynamic C Program:

```
/*  MOVE RIGHT  */

/*********************************GLOBAL CONSTANTS*********************/
#define DELAYONE                      500           //Repelling Time
#define DELAYTWO                      300           //Settling Time
#define DELAYTHREE            200           //Check for Settle
#define DELAYSLAVEONE  300            //Should be sighly less than DELAYONE
#define DELAYSLAVETWO  500            //Should adjust ONE+TWO to be ONE+TWO

#define ENBL          1 //***should be 0
#define DISE          0       //***should be 1
#define ATTR          1 //***should be 1
#define REPL          1 //***should be 0

/*********************************GLOBAL VARIABLES*********************/
int message;                          //
int Side_Matrix[12];                                //
int Side_Matrix_A[12];                      //
int move;                                                           //
char master;                                                //
int bit_e;                                                          //
int bit_p;                                                          //
int magnet;                                                         //

/*********************************INIT FUNCTIONS*********************/
void even_master(int);
void odd_master(int);
void even_slave(int);
void odd_slave(int);
void SEND_MESSAGE_PREPARE(void);
void Write_Ports(int, int);
void SEND_MESSAGE_START_STOP(int);
void get_side_matrix(void);
void setup_bits(char, int);



//************************************************************************
//************************************************************************

int main(void)  {


/********************************INTI VARS******************************************/

               int Phase;                                  //
               int Enable;                                 //
               int Ack_Response;                   //
               int magnet_number;          //
               int side_number;                    //
               int active_side;                    //Side where other module is attached
               int odd_even;                       //odd or even number

               int zzz,k,m,i,rrr,aaa;  // counters

/********************************INIT VAR VALUES*******************************************/

message=0;
move=0;
active_side=100;
zzz=0;
```

```
master='m';

/*********************************SETUP PORTS*******************************************/

       WrPortI(SPCR, &SPCRShadow, 0x80);                          //Port A

       WrPortI(PBDDR, NULL, 0xF0);                                        //Port B
              BitWrPortI(PBDR, &PBDRShadow, 0, 4);
              BitWrPortI(PBDR, &PBDRShadow, 0, 5);
              BitWrPortI(PBDR, &PBDRShadow, 0, 6);
              BitWrPortI(PBDR, &PBDRShadow, 0, 7);

       BitWrPortI(PCFR, &PCFRShadow, 0, 0);                //Port C
       BitWrPortI(PCFR, &PCFRShadow, 0, 2);                //Port C
       BitWrPortI(PCFR, &PCFRShadow, 0, 4);                //Port C
              BitWrPortI(PCDR, &PCDRShadow, 0, 0);
              BitWrPortI(PCDR, &PCDRShadow, 0, 2);
              BitWrPortI(PCDR, &PCDRShadow, 0, 4);


       WrPortI ( PDFR,NULL,0x00 );                                        //Port D
              WrPortI ( PDDCR,NULL,0x00 );
                     WrPortI ( PDDDR,NULL,0xFF );
                            WrPortI ( PDDR,NULL,0x00 );


       WrPortI(PEFR,NULL, 0x00);                                                 //Port E
              WrPortI(PEDDR, NULL, 0x00);

       WrPortI(PFFR, NULL, 0x00);                                                //Port F
       WrPortI(PFDCR, NULL, 0x00);
                     WrPortI(PFDDR, NULL, 0x00);

       WrPortI(PGFR,NULL, 0x00);                                                 //Port G
              WrPortI(PGDCR, NULL, 0x00);
                     WrPortI(PGDDR, NULL, 0xFF);
                            WrPortI(PGDR, NULL, 0x00);


/*********************************BEGIN ENDLESS LOOP*************************************/

while(1) {

//*************************************************************************************
//*************************************************************************************
//*************************************************************************************

              costate {
                     runwatch();
              }//END COSTATE

//*************************************************************************************
//*************************************************************************************
//*************************************************************************************

              costate {
              //read in hull effect sensors to side matrix

                     get_side_matrix();                       //Get sensor values

                     for (rrr=0; rrr<=11; rrr++){
                            if (Side_Matrix[rrr]==1)
       //Record the active side
                                          active_side=rrr;
                            }// END FOR

              } //END COSTATE
```

```
//***************************************************************************************
//***************************************************************************************
//***************************************************************************************

               costate {
       // Look for signal to move right

                       if(BitRdPortI(PFDR,0)==1){            //      Signal on Port F bit 0 switch
                               waitfor(DelayMs(200));                // Wait
                                       if(BitRdPortI(PFDR,0)==1){    // Check Port F bit 0 again
                                               master='m';
       // Make Master Module
                                               move=1;
       // Signal a move
                                       }//END IF
                       }// END IF
               }// END COSTATE

//***************************************************************************************
//***************************************************************************************
//***************************************************************************************

               costate {
       // Look for signal to from other processor

                       if (BitRdPortI(PFDR,1)==1){   //      Signal on Port F bit 1 from master
                               master='s';
       // Become the Slave
                               move=1;
       // Signal a move
                       }//END IF

               }//END COSTATE

//***************************************************************************************
//***************************************************************************************
//***************************************************************************************

               costate {

                       if (move==1 && master=='m') { //If Master - Wait for a move to be signaled

                               if (active_side==100)  // If no side is found break out of the loop
                                       break;

                               if ((active_side+2)%2>0){                    // See if side is Odd
                                       master='s';                   // Become the Slave
                                       odd_slave(active_side);               // Run Odd as Slave
                               }//END IF

                               else {                                // See if side is Even
                                       even_master(active_side);             // Run Even as Master
                               }//END ELSE

                       } // END IF
               }// END COSTATE

//***************************************************************************************
//***************************************************************************************
//***************************************************************************************

               costate {

                       if (move==1 && master=='s') { // If Slave - wait for a move to be signaled

                               if (active_side==100)  / If no side is found break out of the loop
```

```
                                    break;

                            if ((active_side+2)%2>0){                   // See if side is Odd
                                    master='m';                 // Become the Master
                                    odd_master(active_side);            // Run Odd as Master
                            }//END IF

                            else {                      // See if side is Even
                                    even_slave(active_side);        // Run Even as Slave
                    }//END ELSE

                    } // END IF
            }// END COSTATE

//*************************************************************************************
//*************************************************************************************
//*************************************************************************************


        } //END LOOP WHILE 1

return(0);                  // RETURN 0

} // END MAIN


//*************************************************************************************
//*************************************************************************************

void even_master(int side){

        runwatch();

        while (move==1){                            //while no move has occured run this loop

//************************************************************************************
//************************************************************************************
//*************************************************************************************

                costate {

                            if(master='m'){

                                    setup_bits('s', side); // setup bits for first magnet

                                    SEND_MESSAGE_PREPARE();         // COMMS

                                    Write_Ports(REPL,ENBL);          // enable repelling

                                    waitfor(DelayMs(DELAYONE));     / DELAY defined atop

                                    Write_Ports(REPL,DISE);         // disable repelling

                                    magnet=magnet+1;                // move to next magnet

                                    setup_bits('m', magnet);        // setup bits for next magnet

                                    Write_Ports(ATTR,ENBL);         // enable attraction

                                    waitfor(DelayMs(DELAYTWO));     // DELAY defined atop

                            }//END IF MASTER

                            if (side==11)
                                    side=-1;
```

```
                              waitfor(Side_Matrix[side+1]==1); //Check to see if in new state is
the desired one

                              waitfor(DelayMs(DELAYTHREE));

                                     if (Side_Matrix[side+1]==1){
                                            Write_Ports(ATTR,DISE);
                                            move=0;
                                            break;
                                            } //END IF SIDE INCREASE AND DELAY

                                     else
                                            Write_Ports(ATTR,DISE);
                                            move=0;
                       //TEMP
/**************        troubleshootmove();    *****/

                       } //END COSTATE

//*************************************************************************************
//*************************************************************************************

                       costate{
                                     get_side_matrix();
                       }//END COSTATE

//**************************************************************************************
//**************************************************************************************

       }      //END WHILE LOOP
} //END FUNCTION




//*************************************************************************************
//*************************************************************************************


void get_side_matrix(void) {
       //Get the Sensor Values for the sides
                       int pqp,rqr;

                       for (pqp=0; pqp<=7; pqp++){

                              Side_Matrix[pqp]=BitRdPortI(PADR,pqp);
       //read PP A 0-7 to side matrix[0-7]
                              runwatch();
                              }
                       for (pqp=8; pqp<=11; pqp++) {
                              if (pqp==9)
                                     Side_Matrix[pqp]=BitRdPortI(PCDR,1);                //read
PP C 1 to side matrix[9]
                              else

                                     Side_Matrix[pqp]=BitRdPortI(PBDR,pqp-8);            //read
PP B 0,2,3 to side matrix[8,10,11]
                              }

                       for (rqr=0; rqr<=11; rqr++)
                              Side_Matrix_A[rqr]=Side_Matrix[rqr];                       //Store
side matrix to side_matrix_a

}// END FUNCTION
```

```
//*************************************************************************************
//*************************************************************************************
//*************************************************************************************


void setup_bits(char side_magnet, int value){

if(side_magnet=='s'){                        // FOR A SIDE INPUT
        switch (value) {                                    // Setup Bits

                case 0:                                             // Sides 11-0
                case 11:
                        magnet=6;                                   // Is magnet 6
                        bit_e=2;                                            // enable bit is 2 on
Port C
                        bit_p=5;                                            // phase  bit is 5 on
Port D
                        break;
                case 1:                                             // Sides 1-2
                case 2:
                        magnet=1;                                   // Is magnet 1
                        bit_e=4;                                            // enable bit is 4 on
Port B
                        bit_p=0;                                            // phase  bit is 0 on
Port D
                        break;
                case 3:                                             // Sides 3-4
                case 4:
                        magnet=2;                                   // Is magnet 2
                        bit_e=5;                                            // enable bit is 5 on
Port B
                        bit_p=1;                                            // phase  bit is 1 on
Port D
                        break;
                case 5:                                             // Sides 5-6
                case 6:
                        magnet=3;                                   // Is magnet 3
                        bit_e=6;                                            // enable bit is 6 on
Port B
                        bit_p=2;                                            // phase  bit is 2 on
Port D
                        break;
                case 7:                                             // Sides 7-8
                case 8:
                        magnet=4;                                   // Is magnet 4
                        bit_e=7;                                            // enable bit is 7 on
Port B
                        bit_p=3;                                            // phase  bit is 3 on
Port D
                        break;
                case 9:                                             // Sides 9-10
                case 10:
                        magnet=5;                                   // Is magnet 5
                        bit_e=0;                                            // enable bit is 0 on
Port C
                        bit_p=4;                                            // phase  bit is 4 on
Port D
                        break;
                default:
                        magnet=-1;
        }//END SWITCH
}//END IF

if(side_magnet=='m'){                        //FOR A MAGNET INPUT
        switch (value) {                                    //setup bits
```

```
               case 0:                                             // For Magnet 6
               case 6:
                      bit_e=2;                                            // enable bit is 2 on
Port C
                      bit_p=5;                                            // phase  bit is 5 on
Port D
                      break;
               case 7:                                             // For Magnet 1
               case 1:
                      magnet=1;
                      bit_e=4;                                            // enable bit is 4 on
Port B
                      bit_p=0;                                            // phase  bit is 0 on
Port D
                      break;
               case 2:                                             // For Magnet 2
                      bit_e=5;                                            // enable bit is 5 on
Port B
                      bit_p=1;                                            // phase  bit is 1 on
Port D
                      break;
               case 3:                                             // For Magnet 3
                      bit_e=6;                                            // enable bit is 6 on
Port B
                      bit_p=2;                                            // phase  bit is 2 on
Port D
                      break;
               case 4:                                             // For Magnet 4
                      bit_e=7;                                            // enable bit is 7 on
Port B
                      bit_p=3;                                            // phase  bit is 3 on
Port D
                      break;
               case 5:                                             // For Magnet 5
                      bit_e=0;                                            // enable bit is 0 on
Port C
                      bit_p=4;                                            // phase  bit is 4 on
Port D
                      break;
               default:
                      magnet=-1;
        }//END SWITCH
}//end if
}//END FUNCTION


//*************************************************************************************
//*************************************************************************************
//*************************************************************************************

void SEND_MESSAGE_PREPARE(void){
int xx;
xx=0;
       while(xx==0){
       costate{
       BitWrPortI(PCDR, &PCDRShadow, 1, 4);
              waitfor(DelayMs(100));
       BitWrPortI(PCDR, &PCDRShadow, 0, 4);
       xx=1;
       }
       }
}// END FUNCTION
```

```
//*************************************************************************************
//*************************************************************************************

void Write_Ports(int phase, int enable){

       BitWrPortI(PDDR, &PDDRShadow, phase, bit_p);

       if (magnet<5)
              BitWrPortI(PBDR, &PBDRShadow, enable, bit_e); // Enable the repelling magnet if a
b port
       else
              BitWrPortI(PCDR, &PCDRShadow, enable, bit_e); // Enable the repelling magnet if a
c port

} //END FUNCTION


//*************************************************************************************
//*************************************************************************************

void odd_slave(int side){

       runwatch();

       while (move==1){                                    //while no move has occured run this
loop

//*************************************************************************************
//*************************************************************************************
//**************************************************************************************


              costate {

                            if(master='s'){

                                   setup_bits('s', side);
              // setup bits magnet

                                   SEND_MESSAGE_PREPARE();                      //Tell other
module to engage as master

                                   waitfor(DELAYSLAVEONE);
              // SLAVE DELAY ONE defined atop

                                   Write_Ports(ATTR,ENBL);
                     // enable attraction

                                   waitfor(DelayMs(DELAYSLAVETWO));
       // SLAVE DELAY TWO defined atop


                            }//END IF SLAVE

                            if (side==11)
                                   side=-1;

                            waitfor(Side_Matrix[side+1]==1);                           //Check
to see if in new state is the desired one

                            waitfor(DelayMs(DELAYTHREE));

                                   if (Side_Matrix[side+1]==1){
                                          Write_Ports(ATTR,DISE);
                                          move=0;
                                          break;
```

```
                                        } //END IF SIDE INCREASE AND DELAY

                                else
                                        Write_Ports(ATTR,DISE);
                                        move=0;
        // TEMP
/**************         troubleshootmove();     *****/

                        } //END COSTATE

//**************************************************************************************
//*************************************************************************************
//*************************************************************************************

                        costate{
                                get_side_matrix();
                        }//END COSTATE

//**************************************************************************************
//**************************************************************************************
//**************************************************************************************

        }       //END WHILE LOOP
} //END FUNCTION

//**************************************************************************************
//**************************************************************************************


void even_slave(int side){

        runwatch();

        while (move==1){                                        //while no move has occured run this
loop

//*************************************************************************************
//*************************************************************************************
//**************************************************************************************

                costate {

                        if(master='s'){

                                setup_bits('s', side);
                // setup bits magnet

                                waitfor(DELAYSLAVEONE);
                // SLAVE DELAY ONE defined atop

                                Write_Ports(ATTR,ENBL);
                        // enable attraction

                                waitfor(DelayMs(DELAYSLAVETWO));
        // SLAVE DELAY TWO defined atop


                        }//END IF SLAVE

                        if (side==0)
                                side=12;

                        waitfor(Side_Matrix[side-1]==1);                                //Check
to see if in new state is the desired one

                        waitfor(DelayMs(DELAYTHREE));
```

```
                                      if (Side_Matrix[side-1]==1){
                                              Write_Ports(ATTR,DISE);
                                              move=0;
                                              break;
                                              } //END IF SIDE DECREASE AND DELAY

                                      else
                                              Write_Ports(ATTR,DISE);
                                              move=0;
              //TEMP
/**************       troubleshootmove();    *****/

                      } //END COSTATE

//*****************************************************************************************
//****************************************************************************************
//****************************************************************************************

                      costate{
                                      get_side_matrix();
                      }//END COSTATE

//*****************************************************************************************
//*****************************************************************************************
//*****************************************************************************************

       }      //END WHILE LOOP
} //END FUNCTION


//*****************************************************************************************
//*****************************************************************************************


void odd_master(int side){

       runwatch();

       while (move==1){                                   //while no move has occured run this
loop

//*****************************************************************************************

              costate {

                              if(master='m'){

                                      setup_bits('s', side);
              // setup bits for first magnet


                                      Write_Ports(REPL,ENBL);
                      // enable repelling

                                      waitfor(DelayMs(DELAYONE));
       // DELAY defined atop

                                      Write_Ports(REPL,DISE);
                      // disable repelling

                                      magnet=magnet-1;
                              // move to next magnet

                                      setup_bits('m', magnet);
              // setup bits for next magnet
```

```
                                      Write_Ports(ATTR,ENBL);
                       // enable attraction

                                      waitfor(DelayMs(DELAYTWO));
        // DELAY defined atop

                              }//END IF MASTER

                              if (side==0)
                                      side=12;

                              waitfor(Side_Matrix[side-1]==1);                            //Check
to see if in new state is the desired one

                              waitfor(DelayMs(DELAYTHREE));

                                      if (Side_Matrix[side-1]==1){
                                              Write_Ports(ATTR,DISE);
                                              move=0;
                                              break;
                                              } //END IF SIDE INCREASE AND DELAY

                                      else
                                              Write_Ports(ATTR,DISE);
                                              move=0;
                       //TEMP

/**************        troubleshootmove();    *****/

                       } //END COSTATE

//**************************************************************************************
//**************************************************************************************

                       costate{
                                      get_side_matrix();
                       }//END COSTATE

//***************************************************************************************
//***************************************************************************************

        }       //END WHILE LOOP
} //END FUNCTION
```

Appendix B

Final Dynamic C Program:

```
/*********************************GLOBAL CONSTANTS*********************/
#define DELAYONE                      500           //Repelling Time
#define DELAYTWO                      300           //Settling Time
#define DELAYTHREE             200            //Check for Settle
#define DELAYSLAVEONE  300            //Should be slightly less than DELAYONE
#define DELAYSLAVETWO  500            //Should adjust ONE+TWO to be ONE+TWO

#define ENBL          1 //***should be 1
#define DISE          0       //***should be 0
#define ATTR          1 //***should be 1
#define REPL          0 //***should be 0

#define ENBONE                0  //Enable bit numbers
#define ENBTWO                1
#define ENBTHREE              3
#define ENBFOUR               4

#define PHSONE                4  //Phase bit numbers
#define PHSTWO                5
#define PHSTHREE              6
#define PHSFOUR               7

#define SONE                  0       //Sensor Side ports
#define STWO                  1
#define STHREE                2
#define SFOUR                 3
#define SFIVE                 4
#define SSIX                  5
#define SSEVEN                6
#define SEIGHT                7

#define BINBUFSIZE  63
#define BOUTBUFSIZE 63
#define TIMEOUT 20UL   // will time out 20 milliseconds after receiving any
                       // character unless MAXSIZE characters are received

#define MAXSIZE  128
#define MAXSIZE2 160

/*********************************GLOBAL VARIABLES*********************/
int message;                          //
int Side_Matrix[8];                                  // matrix of the sides (active or nonactive)
int Active_Matrix[8];                        // matrix of where the active sides are (in numeric
order)
float position[2];
int rposition;
int number_of_active;                        //      number of active sides
int move;                                                      //      Variable =1 when a
move is occurring
char master;                                           //      Variable for who is
activating m or s
char direction;                                        // Variable for clockwise or
counterclockwise cc or cw
int bit_e;                                                     // The enable bit port #
int bit_p;                                                     // The phase bit port #
int magnet;                                                    // The number of the
electromagnet (1-4)
int movetype;                                          // The type of move, dependant on
activeside and other modules
int comside;
int comport;
int other_side_matrix[8];


/*********************************INIT FUNCTIONS*********************/
void even_master(int);
void odd_master(int);
void even_slave(int);
void odd_slave(int);
void SEND_MESSAGE_PREPARE(void);
void Write_Ports(int, int);
void SEND_MESSAGE_START_STOP(int);
void get_side_matrix(void);
void setup_bits(int);
void get_active_sides(void);
void moving(void);
void type_one_cc(void);
void type_one_cw(void);
void Recieve_Message(void);
void get_com_side(void);
void communicate(int, int);
void set_comport(void);
void reset_com(void);


//*************************************************************************************
//*************************************************************************************

int main(void)  {


/*********************************INTI VARS********************************************/

               int Phase;                                 //
               int Enable;                                //
               int Ack_Response;                   //
               int magnet_number;           //
               int side_number;                    //
               int active_side;                    //Side where other module is attached
               int odd_even;                       //odd or even number

               int zzz,k,m,i,rrr,aaa;  // counters

/*********************************INIT VAR VALUES*******************************************/

message=0;
move=0;
active_side=100;
zzz=0;
master='m';
movetype=0;


/*********************************SETUP PORTS*******************************************/

       WrPortI(SPCR, &SPCRShadow, 0x80);                         //Port A

       WrPortI(PBDDR, NULL, 0x01);                                       //Port B
       BitWrPortI(PBDR, &PBDRShadow, 1, 0);                //put reset serail low


                                             //PORT C  is serial coms


       WrPortI ( PDFR,NULL,0x00 );                                       //Port D
              WrPortI ( PDDCR,NULL,0x00 );
                     WrPortI ( PDDDR,NULL,0xFF );
                            WrPortI ( PDDR,NULL,0x00 );


       WrPortI(PEFR,NULL, 0x00);                                                //Port E
              WrPortI(PEDDR, NULL, 0xF0);
                     WrPortI(PEDR, NULL, 0xF0);
       WrPortI(PFFR, NULL, 0x00);                                                  //Port F
       WrPortI(PFDCR, NULL, 0x00);
                      WrPortI(PFDDR, NULL, 0xFF);
                                     WrPortI(PFDR, NULL, 0x00);

       WrPortI(PGFR,NULL, 0x00);                                                   //Port G
              WrPortI(PGDCR, NULL, 0x00);
                      WrPortI(PGDDR, NULL, 0xFF);
                             WrPortI(PGDR, NULL, 0x00);


/*********************************BEGIN ENDLESS LOOP*************************************/


BitWrPortI(PBDR, &PBDRShadow, 0, 0);                //put reset serial high


while(1) {

//*************************************************************************************
//*************************************************************************************
//*************************************************************************************

              costate {
                      runwatch();
              }//END COSTATE

//*************************************************************************************
//*************************************************************************************
//*************************************************************************************

              costate {
              //read in hull effect sensors to side matrix

                      get_side_matrix();                        //Get sensor values
                      get_active_sides();

              } //END COSTATE

//*************************************************************************************
//*************************************************************************************
//*************************************************************************************

              costate {
       // Look for signal to move right

                      if(BitRdPortI(PBDR,2)==1 || BitRdPortI(PBDR,3)==1 || BitRdPortI(PBDR,4)==1
|| BitRdPortI(PBDR,5)==1 || BitRdPortI(PBDR,6)==1 || BitRdPortI(PBDR,7)==1 ||
BitRdPortI(PEDR,6)==1 || BitRdPortI(PEDR,7)==1){                           //      Signal
on Port F bit 0 switch
                             Recieve_Message();
       // Get the message
                      }// END IF
              }// END COSTATE

//*************************************************************************************
//*************************************************************************************
//*************************************************************************************

              costate {

                      if (move==1){          // If Master - Wait for a move to be signaled
                             moving();

                             if (movetype==1 & direction=='c')
                                    type_one_cc();
```

```
                              if (movetype==1 & direction=='w')
                                      type_one_cw();
                      } // END IF


              }// END COSTATE

//*****************************************************************************************
//*****************************************************************************************
//*****************************************************************************************



       } //END LOOP WHILE 1

return(0);                 // RETURN 0

} // END MAIN


//*****************************************************************************************
//*****************************************************************************************

void get_active_sides(){
       int gas;
       int gas1;
       number_of_active=0;
       gas1=0;

   for (gas=0; gas<=7; gas++){
                                      Active_Matrix[gas]=0;
                              }//END FOR

       for (gas=0; gas<=7; gas++){
                              if(Side_Matrix[gas]==0){                            //side matrix
for active sides
                                      number_of_active++;
                                      Active_Matrix[gas1+1]=gas;
                                      gas1++;
                                      }                      //END IF
                              }                      //END FOR


}  //END FUNCTION


//*****************************************************************************************
//*****************************************************************************************
//*****************************************************************************************

void moving(void){

       if (number_of_active==100)                         // If no active side is found break
out of the loop
                      move=0;

       if(direction=='c' & move==1){                      // IF conuterclockwise movement

                              if ((Active_Matrix[0]+2)%2>0 && Active_Matrix[0]>1){
       // See if side is Odd
                                              master='s';
               // SLAVE
                                              movetype=1;
          // Type 2 move
                              }//END IF
                              if ((Active_Matrix[0]+2)%2==1 && Active_Matrix[1]==2){
       // See if side is Odd
                                             master='s';
              // SLAVE
                                             movetype=1;
          // Type 2 move
                              }//END IF

                              if (master!='s') {
              // All other case
                                             master='m';
       //MASTER
                                             movetype=1;                                  // Move
Type 1
                              }//END IF

       } // END IF

       if (direction=='w' & move==1) {                             // If clockwise movement

                              if ((Active_Matrix[0]+2)%2>0 && Active_Matrix[0]>1){
       // See if side is Odd
                                      master='m';
       // MASTER
                                      movetype=1;
       // Move Type 1
                              }//END IF

                              if ((Active_Matrix[0]+2)%2==1 && Active_Matrix[1]==2){
       // See if side is Odd
                                      master='m';
       // MASTER
                                      movetype=1;
// Type 1 move
                              }//END IF

                              if (master!='m') {                                          // See
if side is Even
                                      master='s';
       // SLAVE
                                      movetype=1;
       // Type 1 move
                              }//END IF

       } // END IF

}// END FUNCTION

//*************************************************************************************
//*************************************************************************************


void type_one_cc(void){

       runwatch();

       while (move==1){                                    //while no move has occured run this
loop

//************************************************************************************
//************************************************************************************
//*************************************************************************************

              costate {
```

```
                                if(master='m'){

                                        if(Active_Matrix[0]==1)
                                                magnet=4;
                                        else
                                                magnet=Active_Matrix[0]/2;

                                        setup_bits(magnet);
        // setup bits for first magnet

                                        Write_Ports(REPL,ENBL);
                        // enable repelling

                                        waitfor(DelayMs(DELAYONE));
        // DELAY defined atop

                                        Write_Ports(REPL,DISE);
                        // disable repelling

                                        magnet=magnet+1;
                                // move to next magnet

                                        if (magnet==5)
                                                magnet=1;

                                        setup_bits(magnet);
        // setup bits for next magnet

                                        Write_Ports(ATTR,ENBL);
                        // enable attraction

                                        waitfor(DelayMs(DELAYTWO));
        // DELAY defined atop

                                }//END IF MASTER

                                if (Active_Matrix[1]==8)
                                        Active_Matrix[1]==1;

                                while(move==1){

                                        waitfor(Side_Matrix[Active_Matrix[1]]==0);
        //Check to see if in new state is the desired one

                                        waitfor(DelayMs(DELAYTHREE));

                                                if (Side_Matrix[Active_Matrix[1]]==0){
                                                        Write_Ports(ATTR,DISE);
                                                        move=0;
                                                        break;
                                                        } //END IF SIDE INCREASE AND DELAY
                                }//END WHILE



                        } //END COSTATE

//*************************************************************************************
//*************************************************************************************

                        costate{
                                        get_side_matrix();
                        }//END COSTATE

                        costate{
                                        if (master='s'){
                                        }//END IF

                                } //END COSTATE
//**************************************************************************************
//**************************************************************************************

        }       //END WHILE LOOP
} //END FUNCTION




//*************************************************************************************
//*************************************************************************************

void Write_Ports(int phase, int enable){

        BitWrPortI(PFDR, &PFDRShadow, phase, bit_p);

        BitWrPortI(PEDR, &PEDRShadow, enable, bit_e);
} //END FUNCTION



//*************************************************************************************
//*************************************************************************************
//*************************************************************************************



void type_one_cw(void){

        runwatch();

        while (move==1){                                        //while no move has occured run this
loop

//**************************************************************************************

                costate {

                                if(master='m'){

                                        magnet=Active_Matrix[1]/2;

                                        setup_bits(magnet);
                // setup bits for first magnet


                                        Write_Ports(REPL,ENBL);
                        // enable repelling

                                        waitfor(DelayMs(DELAYONE));
        // DELAY defined atop

                                        Write_Ports(REPL,DISE);
                        // disable repelling

                                        magnet=magnet-1;
                                // move to next magnet

                                        if (magnet==-1)
                                                magnet=4;

                                        setup_bits(magnet);
        // setup bits for next magnet
```

```
                                      Write_Ports(ATTR,ENBL);
                       // enable attraction
                                      
                                      waitfor(DelayMs(DELAYTWO));
        // DELAY defined atop

                               }//END IF MASTER

                               if (Active_Matrix[0]==1)
                                      Active_Matrix[0]=9;

                               while(move==1){

                                      waitfor(Side_Matrix[Active_Matrix[0]-2]==0);
        //Check to see if in new state is the desired one

                                      waitfor(DelayMs(DELAYTHREE));

                                              if (Side_Matrix[Active_Matrix[0]-2]==0){
                                                     Write_Ports(ATTR,DISE);
                                                     move=0;
                                                     break;
                                                     } //END IF SIDE INCREASE AND DELAY
                               } // END WHILE

                       } //END COSTATE

//***************************************************************************************
//***************************************************************************************

                       costate{
                                      get_side_matrix();
                       }//END COSTATE

//****************************************************************************************
//****************************************************************************************
                       costate{
                                      if (master='s') {

                                      communicate(0,0);

                                      } //END IF
                       }//END COSTATE


        }       //END WHILE LOOP
} //END FUNCTION


//***************************************************************************************
//***************************************************************************************
//***************************************************************************************

void Recieve_Message(void) {

char mssg;
char data[MAXSIZE];
int n;
int coms;
int q;
coms=1;

get_com_side();
comport=comside;
set_comport();
```

```
serBopen(19200);

serBputc('k');
serBrdFlush();
while ((n = serBread(mssg, MAXSIZE-1, TIMEOUT)) == 0) ;


if (mssg=='a'){                                      //TURN ON A MAGNET
        serBputc('k');
        serBrdFlush();
        while ((n = serBread(data, MAXSIZE-1, TIMEOUT)) == 0) ;
        setup_bits(magnet+data[0]);
                while(q==0){
                        costate{
                                Write_Ports(REPL,ENBL);
                // enable repelling
                                waitfor(DelayMs(DELAYONE));
        // DELAY defined atop
                                Write_Ports(REPL,DISE);
                // disable repelling
                        }//END COSTATE
                }//END WHILE

} //END IF

if (mssg=='b'){                              //  RECIEVE A POSITION
        serBputc('k');
        serBrdFlush();
        while ((n = serBread(data, MAXSIZE-1, TIMEOUT)) == 0) ;
        position[0]=data[0];
        position[1]=data[1];
} //END IF

if (mssg=='c'){                              //  VERIFY A CONNECTION
        serBputc('k');
        serBrdFlush();
        while ((n = serBread(data, MAXSIZE-1, TIMEOUT)) == 0) ;
        reset_com();
        comport=comport+data[0];

        if (comport==9)
                comport=1;
        if (comport==0)
                comport=8;

        set_comport();
        serBputs(7);
        serBrdFlush();
} // END IF

if (mssg=='d'){                              //  SEND A POSITION REQUESTED

   sprintf(position, "%f%d", position, rposition[comside-1]);
        serBputs(position);
        serBrdFlush();
        while ((n = serBread(data, MAXSIZE-1, TIMEOUT)) == 0) ;

//      if (data[0] !=7)
//              ERROR();

} //END IF



serBrdFlush();
serBclose();
reset_com();
```

```
}//END COMS FUNCTION


//*************************************************************************************
//*************************************************************************************

void communicate(int type, int mes){



}//END FUNCTION


//*************************************************************************************
//*************************************************************************************
//*************************************************************************************


void reset_com(void){
int poi;
poi=0;
BitWrPortI(PBDR, &PBDRShadow, 1, 0)

while (poi==0){
        costate{
        waitfor(delayMS(10));
        poi=1;
        }
        }

BitWrPortI(PBDR, &PBDRShadow, 0, 0)
}//END FUNCTION

//*************************************************************************************
//*************************************************************************************
void set_comport(void){
if (comport==1)
        BitWrPortI(PDDR, &PDDRShadow, 1, 0)
if (comport==2)
        BitWrPortI(PDDR, &PDDRShadow, 1, 1)
if (comport==3)
        BitWrPortI(PDDR, &PDDRShadow, 1, 2)
if (comport==4)
        BitWrPortI(PDDR, &PDDRShadow, 1, 3)
if (comport==5)
        BitWrPortI(PDDR, &PDDRShadow, 1, 4)
if (comport==6)
        BitWrPortI(PDDR, &PDDRShadow, 1, 5)
if (comport==7)
        BitWrPortI(PDDR, &PDDRShadow, 1, 6)
if (comport==9)
        BitWrPortI(PDDR, &PDDRShadow, 1, 7)


}//END FUNCTION



//*************************************************************************************
//*************************************************************************************
void get_com_side(void){

if (BitRdPortI(PBDR,2)==1)
        comside=1;
if (BitRdPortI(PBDR,3)==1){
        comside=2;
```

```
       magnet=1;
       }
if (BitRdPortI(PBDR,4)==1)
       comside=3;
if (BitRdPortI(PBDR,5)==1){
       comside=4;
       magnet=2;
       }
if (BitRdPortI(PBDR,6)==1)
       comside=5;
if (BitRdPortI(PBDR,7)==1){
       comside=6;
       magnet=3;
       }
if (BitRdPortI(PEDR,6)==1)
       comside=7;
if (BitRdPortI(PEDR,7)==1){
       comside=8;
       magnet=4;
       }
}//END FUNCTION
```

Appendix C

Generic Array Logic (GAL) Program:

```
Name       COMSCTRL;
Partno     atf22v10c;
Date       6/5/02;
Device     g22v10;
Designer   Sean Patterson;

/**  Inputs  **/

Pin     1=RESET;       /*reset bit*/
Pin     2=Side8in;             /*Side 8 input*/
Pin     3=Side1in;             /*Side 1 input*/
Pin     4=Side2in;             /*Side 2 input*/
Pin     5=Side3in;       /*Side 3 input*/
Pin     6=Side4in;             /*Side 4 input*/
Pin     7=Side5in;             /*Side 5 input*/
Pin     8=Side6in;             /*Side 6 input*/
Pin     9=Side7in;     /*Side 7 input*/
Pin     10=NULL1;      /*na*/
Pin     11=NULL2;      /*na*/

/**  Outputs  **/

Pin     14=side5out;      /*Side 5 output*/
Pin     15=side4out;      /*Side 4 output*/
Pin     16=side6out;     /*Side 6 output*/
Pin     17=side3out;      /*Side 3 output*/
Pin     18=side2out;      /*Side 2 output*/
Pin     19=side1out;      /*Side 1 output*/
Pin     20=side7out;     /*Side 7 output*/
Pin     21=side8out;      /*Side 8 output*/
Pin     22=data;          /*Serial data*/
Pin     23=NULL3;         /*na*/

/**  Logic Equations  **/
Side1out = Side1in & RESET & !side1out & !side2out & !side3out & !side4out & !side5out &
!side6out & !side7out & !side8out;
Side2out = Side2in & RESET & !side1out & !side2out & !side3out & !side4out & !side5out &
!side6out & !side7out & !side8out;
Side3out = Side3in & RESET & !side1out & !side2out & !side3out & !side4out & !side5out &
!side6out & !side7out & !side8out;
Side4out = Side4in & RESET & !side1out & !side2out & !side3out & !side4out & !side5out &
!side6out & !side7out & !side8out;
Side5out = Side5in & RESET & !side1out & !side2out & !side3out & !side4out & !side5out &
!side6out & !side7out & !side8out;
Side6out = Side6in & RESET & !side1out & !side2out & !side3out & !side4out & !side5out &
!side6out & !side7out & !side8out;
Side7out = Side7in & RESET & !side1out & !side2out & !side3out & !side4out & !side5out &
!side6out & !side7out & !side8out;
Side8out = Side8in & RESET & !side1out & !side2out & !side3out & !side4out & !side5out &
!side6out & !side7out & !side8out;
data= (side1out & Side1in) # (side2out & Side2in) # (side3out & Side3in) # (side4out & Side4in)#
(side5out & Side5in) # (side6out & Side6in) # (side7out & Side7in) # (side8out & Side8in)
```